

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 **L^AT_EX** 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
```

¹O 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 dependências externas encontra-se no ficheiro *package.yaml*.

Pode aceder ao **GHCI** utilizando o comando:

```
stack ghci
```

Garanta que se encontra na pasta mais externa **do projeto**. A primeira vez que correr este comando as dependências externas serão instaladas automaticamente.

Para gerar o PDF, garanta que se encontra na directoria *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,

```
data ExpAr a = X
  | N a
  | Bin BinOp (ExpAr a) (ExpAr a)
  | Un UnOp (ExpAr a)
  deriving (Eq, Show)
```

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

```
data BinOp = Sum
  | Product
  deriving (Eq, Show)
data UnOp = Negate
  | E
  deriving (Eq, Show)
```

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,

`Bin Sum X (N 10)`

designa $x + 10$ na notação matemática habitual.

1. A definição das funções `inExpAr` e `baseExpAr` para este tipo é a seguinte:

```
inExpAr = [X, num_ops] where
  num_ops = [N, ops]
  ops = [bin, Un]
  bin (op, (a, b)) = Bin op a b
  baseExpAr f g h j k l z = f + (g + (h × (j × k) + l × z))
```

Defina as funções `outExpAr` e `recExpAr`, e teste as propriedades que se seguem.

Propriedade [QuickCheck] 1 `inExpAr` e `outExpAr` são testemunhas de um isomorfismo, isto é, `inExpAr · outExpAr = id` e `outExpAr · idExpAr = id`:

```
prop_in_out_idExpAr :: (Eq a) => ExpAr a -> Bool
prop_in_out_idExpAr = inExpAr · outExpAr == id
prop_out_in_idExpAr :: (Eq a) => OutExpAr a -> Bool
prop_out_in_idExpAr = outExpAr · inExpAr == id
```

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.

$$\begin{aligned} &prop_sum_idr :: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ &prop_sum_idr a exp = eval_exp a exp \stackrel{?}{=} sum_idr \textbf{ where} \\ &\quad sum_idr = eval_exp a (Bin Sum exp (N 0)) \\ &prop_sum_idl :: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ &prop_sum_idl a exp = eval_exp a exp \stackrel{?}{=} sum_idl \textbf{ where} \\ &\quad sum_idl = eval_exp a (Bin Sum (N 0) exp) \\ &prop_product_idr :: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ &prop_product_idr a exp = eval_exp a exp \stackrel{?}{=} prod_idr \textbf{ where} \\ &\quad prod_idr = eval_exp a (Bin Product exp (N 1)) \\ &prop_product_idl :: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ &prop_product_idl a exp = eval_exp a exp \stackrel{?}{=} prod_idl \textbf{ where} \\ &\quad prod_idl = eval_exp a (Bin Product (N 1) exp) \\ &prop_e_id :: (Floating a, Real a) \Rightarrow a \rightarrow Bool \\ &prop_e_id a = eval_exp a (Un E (N 1)) \equiv expd 1 \\ &prop_negate_id :: (Floating a, Real a) \Rightarrow a \rightarrow Bool \\ &prop_negate_id a = eval_exp a (Un Negate (N 0)) \equiv 0 \end{aligned}$$

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

$$\begin{aligned} &prop_double_negate :: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ &prop_double_negate a exp = eval_exp a exp \stackrel{?}{=} eval_exp a (Un Negate (Un Negate exp)) \end{aligned}$$

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

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

que se encontra na página 12 expressa como um hilomorfismo² e teste as propriedades:

Propriedade [QuickCheck] 4 A função *optimize_eval* respeita a semântica da função *eval*.

$$\begin{aligned} &prop_optimize_respects_semantics :: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ &prop_optimize_respects_semantics a exp = eval_exp a exp \stackrel{?}{=} optimize_eval a exp \end{aligned}$$

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:

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

²Qual é a vantagem de implementar a função *optimize_eval* utilizando um hilomorfismo em vez de utilizar um catamorfismo com um gene "inteligente"?

³Apesar 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.

```
prop_const_rule :: (Real a, Floating a) => a -> Bool
prop_const_rule a = sd (N a) == N 0

prop_var_rule :: Bool
prop_var_rule = sd X == N 1

prop_sum_rule :: (Real a, Floating a) => ExpAr a -> ExpAr a -> Bool
prop_sum_rule exp1 exp2 = sd (Bin Sum exp1 exp2) == sum_rule where
  sum_rule = Bin Sum (sd exp1) (sd exp2)

prop_product_rule :: (Real a, Floating a) => ExpAr a -> ExpAr a -> Bool
prop_product_rule exp1 exp2 = sd (Bin Product exp1 exp2) == prod_rule where
  prod_rule = Bin Sum (Bin Product exp1 (sd exp2)) (Bin Product (sd exp1) exp2)

prop_e_rule :: (Real a, Floating a) => ExpAr a -> Bool
prop_e_rule exp = sd (Un E exp) == Bin Product (Un E exp) (sd exp)

prop_negate_rule :: (Real a, Floating a) => ExpAr a -> Bool
prop_negate_rule exp = sd (Un Negate exp) == Un Negate (sd exp)
```

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 calculando 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*.

```
prop_congruent :: (Floating a, Real a) => a -> ExpAr a -> Bool
prop_congruent a exp = ad a exp == eval_exp a (sd exp)
```

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 $F X = 1 + X$) 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

$$\begin{aligned} fib\ 0 &= 1 \\ fib\ (n + 1) &= f\ n \end{aligned}$$

⁴Lei (3.94) em [2], página 98.

$$f\ 0 = 1$$

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

Obter-se-á de imediato

$$fib' = \pi_1 \cdot \text{for loop init where}$$

$$\text{loop } (fib, f) = (f, fib + f)$$

$$\text{init} = (1, 1)$$

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* colecionam-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 aulas⁶, 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:

$$f'\ a\ b\ c = \pi_1 \cdot \text{for loop init where}$$

$$\text{loop } (f, k) = (f + k, k + 2 * a)$$

$$\text{init} = (c, a + b)$$

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

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

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

$$cat = \dots \cdot \text{for loop init where } \dots$$

que implemente esta função.

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

$$prop_cat = (\geq 0) \Rightarrow (catdef \equiv cat)$$

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 $\{P_0, \dots, P_N\}$ 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.

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

⁶Secção 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:

```
linear1d :: Q → Q → OverTime Q
linear1d a b = formula a b where
  formula :: Q → Q → Float → Q
  formula x y t = ((1.0 :: Q) - (toQ t)) * x + (toQ t) * y
```

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.

```
type NPoint = [Q]
```

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

```
p2d = [1.2, 3.4]
p3d = [0.2, 10.3, 2.4]
```

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

```
type OverTime a = Float → a
```

O anexo C tem definida a função

```
calcLine :: NPoint → (NPoint → OverTime NPoint)
```

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

```
deCasteljau :: [NPoint] → 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.

```
prop_calcLine_def :: NPoint → NPoint → Float → Bool
prop_calcLine_def p q d = calcLine p q d ≡ zipWithM linear1d p q d
```

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

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

```
prop_bezier_sym :: [[Q]] → Gen Bool
prop_bezier_sym l = all (<Δ) · calc_difs · bezs ($) elements ps where
  calc_difs = (λ(x, y) → zipWith (λw v → if w ≥ v then w - v else v - w) x y)
  bezs t = (deCasteljau l t, deCasteljau (reverse l) (fromQ (1 - (toQ t))))
  Δ = 1e-2
```

3. Corra a função `runBezier` e aprecie o seu trabalho⁷ clicando na janela que é aberta (que contém, a verde, um ponto inicial) 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 ,

$$\text{avg } x = \frac{1}{k} \sum_{i=1}^k x_i \quad (2)$$

onde $k = \text{length } x$. 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 fácil de ver que

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

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 = (avg, length)` 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:*

```
prop_avg :: [Double] → Property
prop_avg = nonempty ⇒ diff ≤ 0.000001 where
  diff l = avg l - (avgLTree · genLTree) l
  genLTree = ([lsplit])
  nonempty = (>[])
```

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.

⁷A 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

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

Os diagramas podem ser produzidos recorrendo à *package* L^AT_EX *xymatrix*, por exemplo:

$$\begin{array}{ccc}
 \mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
 \downarrow \langle g \rangle & & \downarrow id + \langle g \rangle \\
 B & \xleftarrow{g} & 1 + B
 \end{array}$$

B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina⁹, 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:

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

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:

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

⁸Exemplos tirados de [2].

⁹Cf. [2], página 102.

C Código fornecido

Problema 1

```
expd :: Floating a => a -> a
expd = Prelude.exp
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 Catalan¹⁰:

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

Problema 3

Algoritmo:

```
deCasteljau :: [NPoint] -> OverTime NPoint
deCasteljau [] = nil
deCasteljau [p] = p
deCasteljau l = λpt -> (calcLine (p pt) (q pt)) pt where
  p = deCasteljau (init l)
  q = deCasteljau (tail l)
```

Função auxiliar:

```
calcLine :: NPoint -> (NPoint -> OverTime NPoint)
calcLine [] = nil
calcLine (p : x) = g p (calcLine x) where
  g :: (Q, NPoint -> OverTime NPoint) -> (NPoint -> OverTime NPoint)
  g (d, f) l = case l of
    [] -> nil
    (x : xs) -> λz -> concat $ (sequenceA [singl · linear1d d x, f xs]) z
```

2D:

```
bezier2d :: [NPoint] -> OverTime (Float, Float)
bezier2d [] = (0, 0)
bezier2d l = λz -> (fromQ × fromQ) · (λ[x, y] -> (x, y)) $ ((deCasteljau l) z)
```

Modelo:

```
data World = World { points :: [NPoint]
  , time :: Float
  }
initW :: World
initW = World [] 0
```

¹⁰Fonte: [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 toQ [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 fromQ 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 (fromQ x) (fromQ 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" }

```

¹¹Pode 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:

```
instance Arbitrary UnOp where
  arbitrary = elements [Negate, E]
instance Arbitrary BinOp where
  arbitrary = elements [Sum, Product]
instance (Arbitrary a) => Arbitrary (ExpAr a) where
  arbitrary = do
    binop <- arbitrary
    unop <- arbitrary
    exp1 <- arbitrary
    exp2 <- arbitrary
    a <- arbitrary
    frequency · map (id × pure) $ [(20, X), (15, N a), (35, Bin binop exp1 exp2), (30, Un unop exp1)]
infixr 5  $\stackrel{?}{=}$ 
( $\stackrel{?}{=}$ ) :: Real a => a -> a -> Bool
( $\stackrel{?}{=}$ ) x y = (to $_{\mathbb{Q}}$  x) == (to $_{\mathbb{Q}}$  y)
```

Outras funções auxiliares

Lógicas:

```
infixr 0 =>
(=>) :: (Testable prop) => (a -> Bool) -> (a -> prop) -> a -> Property
p => f =  $\lambda a \rightarrow p\ a \Rightarrow f\ a$ 
infixr 0 <=>
(<=>) :: (a -> Bool) -> (a -> Bool) -> a -> Property
p <=> f =  $\lambda a \rightarrow (p\ a \Rightarrow \text{property}\ (f\ a)) \ \&\&\. (f\ a \Rightarrow \text{property}\ (p\ a))$ 
infixr 4  $\equiv$ 
( $\equiv$ ) :: Eq b => (a -> b) -> (a -> b) -> (a -> Bool)
f  $\equiv$  g =  $\lambda a \rightarrow f\ a \equiv g\ a$ 
infixr 4 <=
(<=) :: Ord b => (a -> b) -> (a -> b) -> (a -> Bool)
f <= g =  $\lambda a \rightarrow f\ a <= g\ a$ 
infixr 4  $\wedge$ 
( $\wedge$ ) :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
f  $\wedge$  g =  $\lambda a \rightarrow ((f\ a) \wedge (g\ a))$ 
```

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, diagramas e/ou outras funções auxiliares que sejam necessárias.

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

Problema 1

São dadas:

```
cataExpAr g = g · recExpAr (cataExpAr g) · outExpAr
anaExpAr g = inExpAr · recExpAr (anaExpAr g) · g
hyloExpAr h g = cataExpAr h · anaExpAr g
```

```

eval_exp :: Floating a => a -> (ExpAr a) -> a
eval_exp a = cataExpAr (g_eval_exp a)
optimize_eval :: (Floating a, Eq a) => a -> (ExpAr a) -> a
optimize_eval a = hyloExpAr (gopt a) clean
sd :: Floating a => ExpAr a -> ExpAr a
sd = π2 · cataExpAr sd_gen
ad :: Floating a => a -> ExpAr a -> a
ad v = π2 · 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*.

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

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 = i1 ()
outExpAr (N a) = i2 (i1 a)
outExpAr (Bin op a b) = i2 (i2 (i1 (op, (a, b))))
outExpAr (Un op a) = i2 (i2 (i2 (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 (*Na*).

$$\begin{aligned}
& baseExpAr' g f = baseExpAr id g id f f id f \\
\equiv & \quad \{ \text{De acordo com o raciocínio explicado anteriormente} \} \\
& recExpr f = baseExpAr' id f
\end{aligned}$$

Representando graficamente:

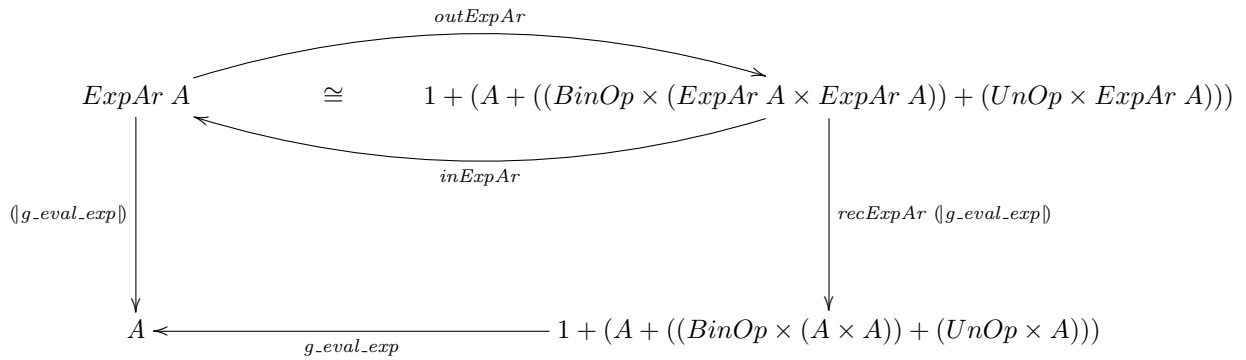
$$\begin{aligned}
& 1 + (A + (BinOp \times (ExpAr A \times ExpAr A) + (UnOp \times ExpAr A))) \\
& \quad \downarrow id + (id + (id \times (f \times f) + (id \times f))) \\
& 1 + (A + (BinOp \times (B \times B) + (UnOp \times B)))
\end{aligned}$$

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 *B*.

$$\begin{aligned}
& baseExpAr' g f = baseExpAr id g id f f id f \\
& -- \\
& recExpAr f = baseExpAr' id f
\end{aligned}$$

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.



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 *a*, respetivamente. (Nota : *a* pertence ao tipo *A*).

$$\begin{aligned}
unAction (Negate, a) &= negate a \\
unAction (E, a) &= expd a \\
binAction (Sum, (a, b)) &= a + b \\
binAction (Product, (a, b)) &= a * b
\end{aligned}$$

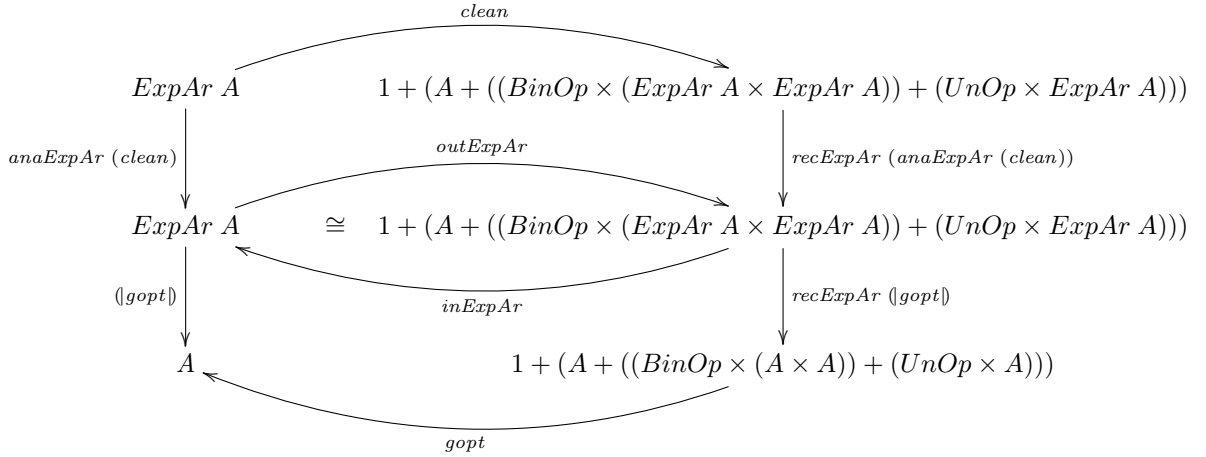
$$g_eval_exp\ a = [\underline{a}, [id, [binAction, unAction]]]$$

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 $clean$.

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

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



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 $ExpAr\ A$ num A . No cálculo realizamos manualmente as operações que envolviam elementos neutros e recorremos à função $binAction$, definida anteriormente, para o caso geral.

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

$$\begin{aligned}
binAction' (Product, (1, b)) &= b \\
binAction' (Product, (a, 1)) &= a \\
binAction' a &= binAction a
\end{aligned}$$

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 sucessivas 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 catamorfismos 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. Utilizamos funções auxiliares responsáveis por derivar cada parte do tipo, tanto em *sd* como em *ad*. Em baixo seguem-se as definições resultantes:

$$\begin{aligned}
sd_gen :: Floating a \Rightarrow & \\
() + (a + ((BinOp, ((ExpAr a, ExpAr a), (ExpAr a, ExpAr a))) + (UnOp, (ExpAr a, ExpAr a)))) \rightarrow & \\
(ExpAr a, ExpAr a) & \\
sd_gen = [derivX, [derivA, [derivBin, derivUn]]] \textbf{ where} & \\
derivX () = (X, N 1) & \\
derivA a = (N a, N 0) & \\
derivBin (Sum, ((a, a'), (b, b'))) = (Bin Sum a b, Bin Sum a' b') & \\
derivBin (Product, ((a, a'), (b, b'))) = (Bin Product a b, Bin Sum (Bin Product a b') (Bin Product a' b)) & \\
derivUn (Negate, (a, a')) = (Un Negate a, Un Negate a') & \\
derivUn (E, (a, a')) = (Un E a, Bin Product (Un E a) a') & \\
\\
ad_gen v = [auxX v, [auxN, [auxBin, auxUn]]] \textbf{ where} & \\
auxX v () = (v, 1) & \\
auxN a = (a, 0) & \\
auxBin (Sum, ((a, b), (c, d))) = ((+) a c, (+) b d) & \\
auxBin (Product, ((a, b), (c, d))) = ((*) a c, (+) ((*) a d) ((*) b c)) & \\
auxUn (Negate, (a, b)) = (-a, -b) & \\
auxUn (E, (a, b)) = (expd a, (*) (expd a) b) &
\end{aligned}$$

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:

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

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

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.

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

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.

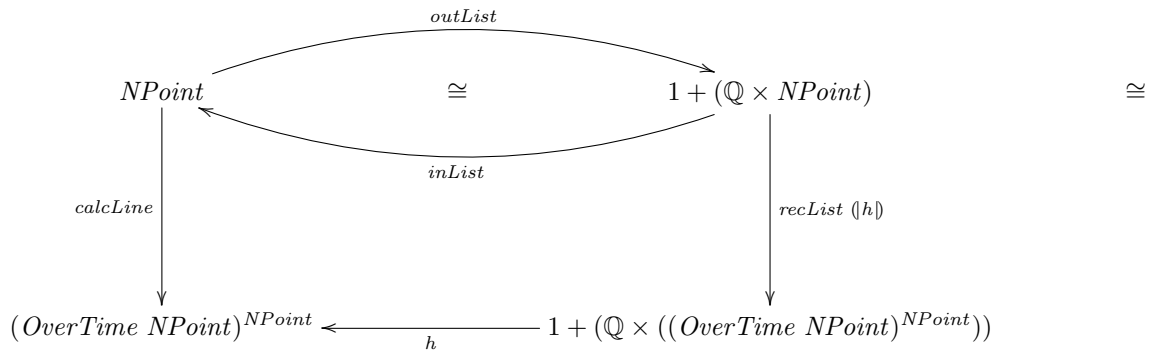
$$\begin{aligned}
c \ 0 &= 1 \\
c \ (n + 1) &= c \ n * d \ n \div e \ (n + 1) \\
d \ 0 &= 2 \\
d \ (n + 1) &= 4 + d \ n \\
e \ 0 &= 2 \\
e \ (n + 1) &= 1 + e \ n \\
cat &= prj \cdot \text{for loop } inic \ \textbf{where} \\
\text{loop } (c, d, e) &= (c * d \div e, 4 + d, 1 + e) \\
inic &= (1, 2, 2) \\
prj \ (c, d, e) &= c
\end{aligned}$$

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

Problema 3

calcLine

Para definirmos a função *calcLine* como um catamorfismo, podemos primeiro representar o diagrama do seu comportamento:



Com este diagrama e com a função já definida no anexo C, compreende-se que, para passar aquela definição para um catamorfismo, temos de construir o gene que nos permita passar do tipo $1 + Q \times (OverTime \ NPoint) \uparrow NPoint$ para $(OverTime \ NPoint) \uparrow NPoint$.

Como no lado esquerdo recebemos um tipo com um único habitante, que é a lista vazia, retornamos *nil*. No lado direito, por sua vez, recebemos um par, com o racional que estava à cabeça da lista original,

e com a aplicação do catamorfismo já feita para a cauda. Assim, tomando como base a definição já dada, podemos definir o $h2$ (lado direito) com a definição idêntica à dada.

```

calcLine :: NPoint → (NPoint → OverTime NPoint)
calcLine = cataList h where
  h = [h1, h2]
  h1 _ _ = nil
  h2 (d, f) l = case l of
    [] → nil
    (x : xs) → λz → concat $ (sequenceA [singl · linear1d d x, f xs]) z

```

deCasteljau

Nesta questão pensamos em transformar o hilomorfismo num catamorfismo seguido de um anamorfismo, no entanto não conseguimos deduzir as definições necessárias para resolver este problema nem efetivamente construir o diagrama, porque não percebemos muito bem o comportamento da função *deCasteljau*.

```

deCasteljau :: [NPoint] → OverTime NPoint
deCasteljau = hyloAlgForm alg coalg where
  coalg = ⊥
  alg = ⊥
  hyloAlgForm = ⊥

```

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.

```

outList' [h] = i1 h
outList' (h : t) = i2 (h, t)
cataList' g = g · recList (cataList' g) · outList'

```

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:

$$\begin{array}{ccc}
A^+ & \xrightarrow{\text{outList}'} & A + (A \times A^+) \\
\downarrow \llbracket [b, q] \rrbracket & & \downarrow \text{recList } \llbracket [b, q] \rrbracket \\
A \times N_0 & \xleftarrow{\llbracket [b, q] \rrbracket} & A + A \times (A \times N_0)
\end{array}$$

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 simplesmente adiciona um ao valor que vem calculado anteriormente. Com isto, definimos as funções desta forma:

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

avgLTree

Solução para árvores de tipo **LTree**:

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

$$\begin{array}{ccc} LTree\ A & \xrightarrow{\text{outLTree}} & A + (LTree\ A \times LTree\ A) \\ \downarrow \llbracket [b, q] \rrbracket & & \downarrow \text{recLTree } \llbracket [b, q] \rrbracket \\ A \times N_0 & \xleftarrow{[b, q]} & A + ((A \times N_0) \times (A \times N_0)) \end{array}$$

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 $\langle id, 1 \rangle$. 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).

$$\begin{aligned} \text{avgLTree} &= \pi_1 \cdot \llbracket g \rrbracket \text{ where} \\ g &= [\langle id, 1 \rangle, \langle \text{auxAvgLTree}, \text{auxLenLTree} \rangle] \\ \text{auxAvgLTree } ((a1, l1), (a2, l2)) &= (((a1 * l1) + (a2 * l2)) / (l1 + l2)) \\ \text{auxLenLTree } ((-, l1), (-, l2)) &= l1 + l2 \end{aligned}$$

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 polytypic

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

- LaTeX, [1](#)
 - [bibtex](#), [2](#)
 - [lhs2TeX](#), [1](#)
 - [makeindex](#), [2](#)
- Cálculo de Programas, [1](#), [2](#), [5](#)
 - Material Pedagógico, [1](#)
 - BTree.hs, [8](#)
 - Cp.hs, [8](#)
 - LTree.hs, [8](#), [14](#)
 - Nat.hs, [8](#)
- Combinador “pointfree”
 - cata*, [8](#), [9](#)
 - either*, [3](#), [8](#)
- Curvas de Bézier, [6](#), [7](#)
- Deep Learning), [3](#)
- DSL (linguagem específica para domínio), [3](#)
- F#, [8](#), [14](#)
- Função
 - π_1 , [6](#), [9](#), [14](#)
 - π_2 , [9](#), [13](#)
 - for*, [6](#), [9](#), [13](#)
 - length*, [8](#)
 - map*, [11](#), [12](#)
 - uncurry*, [3](#)
- Functor, [5](#), [11](#)
- Haskell, [1](#), [2](#), [8](#)
 - Gloss, [2](#), [11](#)
 - interpretador
 - GHCi, [2](#)
 - Literate Haskell, [1](#)
 - QuickCheck, [2](#)
 - Stack, [2](#)
- Números de Catalan, [6](#), [10](#)
- Números naturais (N), [5](#), [6](#), [9](#)
- Programação
 - dinâmica, [5](#)
 - literária, [1](#)
- Racionais, [7](#), [8](#), [10–12](#)
- U.Minho
 - Departamento de Informática, [1](#)

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.