

Cálculo de Programas

Trabalho Prático

MiEI+LCC — 2020/21

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	21
a93270	João Barbosa
a93262	Simão Cunha
a93277	Tiago Silva

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-processor 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* · *inExpAr* = *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.

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

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 algebrá* 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

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

$$loop\ (fib, f) = (f, fib + f)$$

$$init = (1, 1)$$

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:

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

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

$$init = (c, a + b)$$

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

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

que implemente esta função.

Propriedade [QuickCheck] 7 A função proposta coincidem 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.

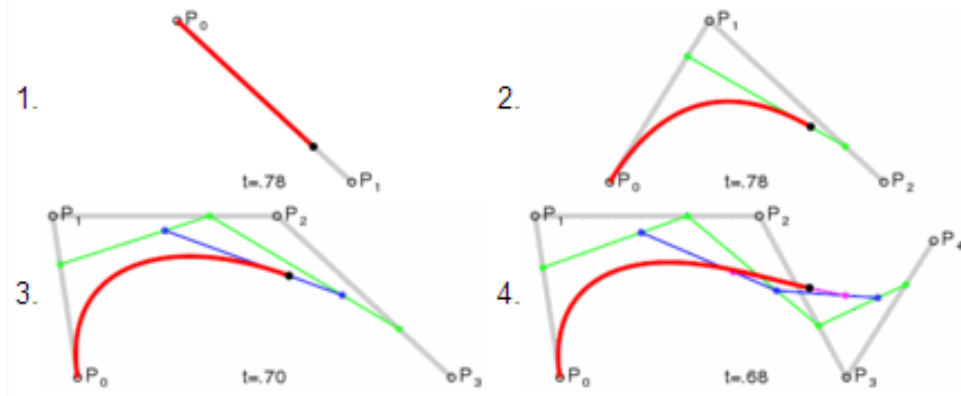


Figure 1: Exemplos de curvas de Bézier retirados da [Wikipedia](#).

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:

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

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 (1):

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

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 -> p a => f a
infixr 0 <=>
(<=>) :: (a -> Bool) -> (a -> Bool) -> a -> Property
p <=> f =  $\lambda$ a -> (p a => property (f a)) .&&. (f a => property (p a))
infixr 4  $\equiv$ 
( $\equiv$ ) :: Eq b => (a -> b) -> (a -> b) -> (a -> Bool)
f  $\equiv$  g =  $\lambda$ a -> f a  $\equiv$  g a
infixr 4  $\leq$ 
( $\leq$ ) :: Ord b => (a -> b) -> (a -> b) -> (a -> Bool)
f  $\leq$  g =  $\lambda$ a -> f a  $\leq$  g a
infixr 4  $\wedge$ 
( $\wedge$ ) :: (a -> Bool) -> (a -> Bool) -> (a -> Bool)
f  $\wedge$  g =  $\lambda$ a -> ((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, 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:

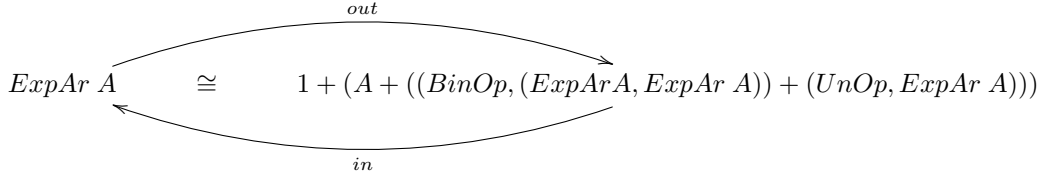
```
cataExpAr g = g · recExpAr (cataExpAr g) · outExpAr
anaExpAr g = inExpAr · recExpAr (anaExpAr g) · g
hyloExpAr h g = cataExpAr h · anaExpAr g
```

```

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)

```

O primeiro problema baseia-se em definir a função “saída” do tipo `ExpAr`. É-nos dado tanto o tipo da função `outExpAr` como o do seu isomorfismo `inExpAr`, sendo assim possível representar o seu diagrama.



Assim, conseguimos perceber de imediato a definição de `outExpAr`:

```

outExpAr :: ExpAr a -> () + (a + ((BinOp, (ExpAr a, ExpAr a)) + (UnOp, ExpAr a)))
outExpAr X = i1 ()
outExpAr (N x) = i2 (i1 x)
outExpAr (Bin op exp1 exp2) = i2 (i2 (i1 (op, (exp1, exp2))))
outExpAr (Un op exp1) = i2 (i2 (i2 (op, exp1)))

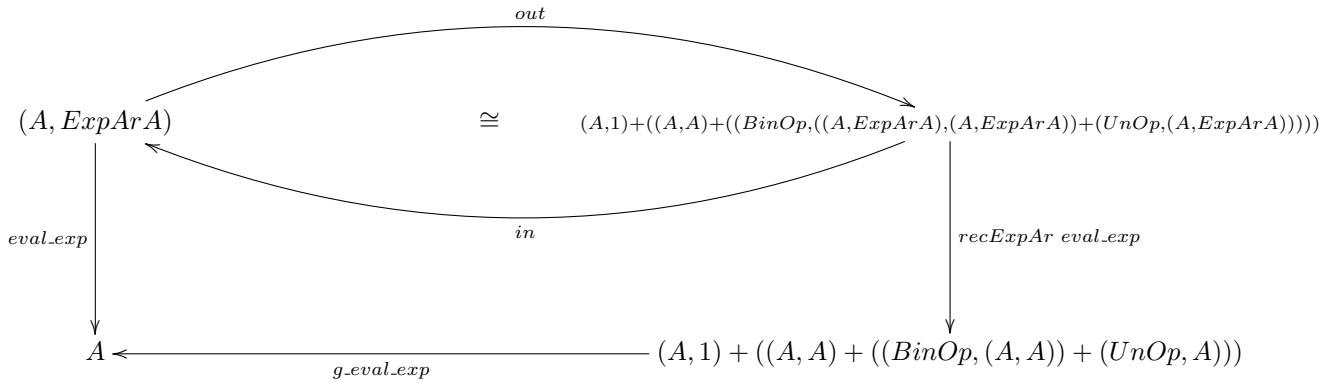
```

Relativamente à função `recExpAr`. Esta recebe uma função `f` e chama a função `baseExpAr`, aplicando esse `f` apenas aos argumentos `ExpAr`, deixando o resto intacto com a função `id`. Fazendo assim a função recursiva do tipo `ExpAr`.

Chegamos à conclusão da expressão de `recExpAr` através da dica dada pelo professor nas FAQ's da página da disciplina (Q9).

```
recExpAr f = baseExpAr id id id f f id f
```

No enunciado é nos dado `cataExpAr` e, com isso, fazer um gene de um catamorfismo que recursivamente calculasse o valor de um `ExpAr`. Começamos por fazer o diagrama, para nos servir de ajuda à realização do gene.



```
recExpAr eval_exp = id + id + id + eval_exp + eval_exp + id + eval_exp
```

Com a ajuda do diagrama chegamos ao seguinte gene:

```

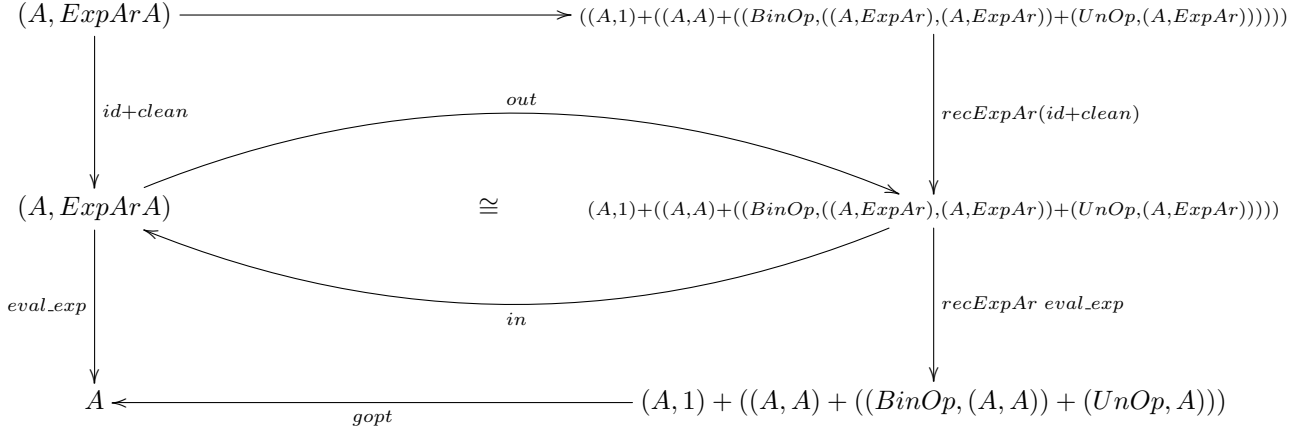
g_eval_exp var = [g_eval_x, [g_eval_na, [g_eval_binop, g_eval_unop]]]
where
  g_eval_x () = var
  g_eval_na b = b
  g_eval_binop (op, (a1, a2))
    | op == Sum = a1 + a2
    | otherwise = a1 * a2
  g_eval_unop (op, a1)

```

| $op \equiv Negate = (-1) * a1$
 | $otherwise = Prelude.exp \ (a1)$

--

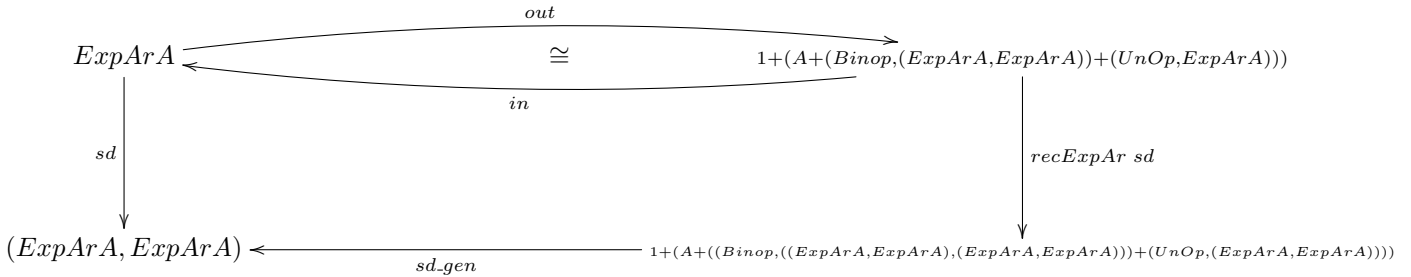
A seguinte questão baseava-se em criar uma solução mais eficiente à questão anterior, e para isso era necessário fazer um hilomorfismo, ou seja, juntar ao catamorfismo já feito, um anamorfismo que realiza-se uma otimização na expressão, utilizando, por exemplo, das propriedades neutras da multiplicação. Sabendo disso, criamos mais uma vez um diagrama para nos ajudar a chegar à solução do problema.



Seguindo o diagrama, chegamos à solução do clean. Como dissemos, a solução do gene *gopt* (gene utilizado no catamorfismo do hilomorfismo), foi reutilizado o gene do catamorfismo da questão anterior, *g_eval_exp*.

```
clean :: (Floating a, Eq a) => ExpAr a -> () + (a + ((BinOp, (ExpAr a, ExpAr a)) + (UnOp, ExpAr a)))
clean (Bin Sum (N 0) b) = outExpAr b
clean (Bin Sum a (N 0)) = outExpAr a
clean (Bin Product (N 0) _) = i2 (i1 0)
clean (Bin Product _ (N 0)) = i2 (i1 0)
clean (Bin Product a (N 1)) = outExpAr a
clean (Bin Product (N 1) b) = outExpAr b
clean (Un Negate (N 0)) = i2 (i1 0)
clean (Un Negate (Un Negate x)) = outExpAr x
clean (Un E (N 0)) = i2 (i1 1)
clean exp = outExpAr exp
gopt var = g_eval_exp var
```

Nesta questão, era pedido que realizássemos uma função que fizesse a derivada de uma expressão, através de um catamorfismo. Percebemos através do enunciado que o gene devolvia um par, e que, a função principal só aproveitava o segundo elemento. Com isso percebemos que o gene colocava a expressão intacta no primeiro elemento do par, e a sua derivada no segundo, para que quando aparecesse uma multiplicação conseguisse completar a derivada, já que esta precisa de parte da expressão antes de ser derivada. Depois de toda a análise, chegamos ao seguinte diagrama:



Através do diagrama, conseguimos concluir o seguinte gene:

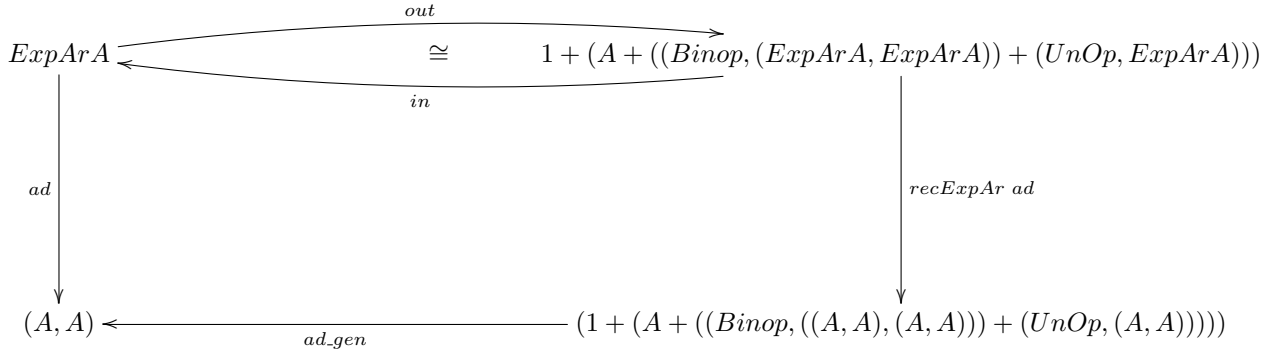
```
sd_gen :: Floating a =>
  () + (a + ((BinOp, ((ExpAr a, ExpAr a), (ExpAr a, ExpAr a))) + (UnOp, (ExpAr a, ExpAr a))))
  -> (ExpAr a, ExpAr a)
sd_gen = [sd_x, [sd_n, [sd_binop, sd_unop]]] where
```

```

sd_x _ = (X, N 1)
sd_n x = (N x, N 0)
sd_binop (op, ((x1, y1), (x2, y2)))
  | op ≡ Sum = ((Bin Sum x1 x2), (Bin Sum y1 y2))
  | otherwise = ((Bin Product x1 x2), (Bin Sum (Bin Product x1 y2) (Bin Product y1 x2)))
sd_unop (op, (x, y))
  | op ≡ Negate = ((Un op x), (Un op y))
  | otherwise = (Un op x, Bin Product (Un op x) y)

```

Esta questão é muito parecida à anterior, só que calculando a derivada com um valor para o x dado. Sendo assim, utilizamos do mesmo raciocínio e fizemos o seguinte diagrama:



Com o diagrama chegamos ao seguinte gene:

```

ad_gen a = [ad_x, [ad_n, [ad_binop, ad_unop]]] where
  ad_x _ = (a, 1)
  ad_n x = (x, 0)
  ad_binop (op, ((x1, y1), (x2, y2)))
    | op ≡ Sum = (x1 + x2, y1 + y2)
    | otherwise = (x1 * x2, (x1 * y2) + (y1 * x2))
  ad_unop (op, (x, y))
    | op ≡ Negate = ((-1) * x, (-1) * y)
    | otherwise = (Prelude.exp (x), y * Prelude.exp (x))

```

Problema 2

Definir

```

loop (f, b, c) = ((f * b) `div` c, 4 + b, 1 + c)
inic = (1, 2, 2)
prj (a, b, c) = a

```

por forma a que

```
cat = prj · for loop inic
```

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

Para uma simplificação de cálculos, decidimos utilizar como fórmula para os números de Catalan:

$$\begin{cases} C_0 = 1 \\ C_{n+1} = \frac{2(2n+1)}{n+2} C_n \end{cases}$$

$$C_n = \frac{(2n)!}{(n+1)!(n)!} \Rightarrow C_{n-1} = \frac{(2(n-1))!}{(n)!(n-1)!}$$

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

$$\Rightarrow C_n = \frac{2(2n-1)}{n+1} C_{n-1}$$

que pode ser deduzida **aqui**.

A expressão para calcular o n-ésimo número de Catalan pode ser traduzida pelas seguintes expressões:

$$\begin{cases} C(0) = 1 \\ C(n+1) = (h(n)/r(n)) * C(n) \end{cases}$$

Chegamos ao resultado da função recursiva h através dos seguintes cálculos.

$$\begin{aligned} \begin{cases} h(0) = 2 \\ h(n) = 2(2n+1) \\ h(n+1) = 2(2(n+1)+1) \end{cases} &\equiv \begin{cases} h(0) = 2 \\ h(n) = 4n+2 \\ h(n+1) = 2(2n+2+1) \end{cases} \equiv \begin{cases} h(0) = 2 \\ h(n) = 4n+2 \\ h(n+1) = 4n+2+4 \end{cases} \equiv \\ \begin{cases} h(0) = 2 \\ h(n) = 4n+2 \\ h(n+1) = h(n)+4 \end{cases} \end{aligned}$$

Chegamos ao resultado da função recursiva r através dos seguintes cálculos.

$$\begin{aligned} \begin{cases} r(0) = 2 \\ r(n) = (n+2) \\ r(n+1) = (n+1)+2 \end{cases} &\equiv \begin{cases} r(0) = 2 \\ r(n) = (n+2) \\ r(n+1) = n+2+1 \end{cases} \equiv \begin{cases} r(0) = 2 \\ r(n) = (n+2) \\ r(n+1) = r(n)+1 \end{cases} \end{aligned}$$

Agora estamos em condições de responder ao enunciado.

A função *init* irá ser constituída por um triplo, onde cada campo vai ser constiuído pelo valor obtido no caso de paragem da função c, h e r, respetivamente.

A função *loop* irá ser constiuída por um triplo, onde em cada campo irá estar a função c, h e r, respetivamente.

Por último, a função *proj* será constituída por um triplo com as funções utilizadas, projetando a função c que contém o resultado pretendido (i.e. o n-ésimo número de Catalan).

Problema 3

```
calcLine :: NPoint → (NPoint → OverTime NPoint)
calcLine = cataList h where
  h = [h1, h2]
  h1 _ = nil
  h2 (h, t) l = case l of
    [] → nil
    (x : xs) → λz → concat $ (sequenceA [singl · linear1d h x, t xs]) z
deCasteljau :: [NPoint] → OverTime NPoint
deCasteljau = hyloAlgForm alg coalg where
  coalg = ⊥
  alg = ⊥
hyloAlgForm = ⊥
```

Problema 4

Antes de descobrir o avg.aux tanto para listas como para LTree, será necessário transformar [b,q] num split de funções para podermos aplicar a lei da troca (Lei 28).

$$\begin{aligned} &[b, q] \\ \equiv &\{ \text{Lei 1 - Natural-id} \} \\ &id \cdot [b, q] \\ \equiv &\{ \text{Lei 8 - Reflexão-X} \} \end{aligned}$$

$$\begin{aligned}
& \langle \pi_1, \pi_2 \rangle \cdot [b, q] \\
\equiv & \{ \text{Lei 20 - Fusão + } \} \\
& [\langle \pi_1, \pi_2 \rangle \cdot b, \langle \pi_1, \pi_2 \rangle \cdot q] \\
\equiv & \{ \text{Lei 9 - Fusão X } \} \\
& [\langle \pi_1 \cdot b, \pi_2 \cdot b \rangle, \langle \pi_1 \cdot q, \pi_2 \cdot q \rangle] \\
\equiv & \{ \text{Lei 28 - Lei da troca } \} \\
& [\langle \pi_1 \cdot b, \pi_1 \cdot q \rangle, \langle \pi_2 \cdot b, \pi_2 \cdot q \rangle] \\
& \square
\end{aligned}$$

Solução para listas não vazias:

$$\begin{aligned}
& \text{avg_aux} = ([b, q]) \\
\equiv & \{ \text{Definição de avg_aux } \} \\
& \langle \text{avg}, \text{length} \rangle = ([b, q]) \\
\equiv & \{ \text{Resultado calculado em cima } \} \\
& \langle \text{avg}, \text{length} \rangle = (\langle [\pi_1 \cdot b, \pi_1 \cdot q], [\pi_2 \cdot b, \pi_2 \cdot q] \rangle) \\
\equiv & \{ \text{Lei 52 - Fokkinga e Functor das listas: } F f = \text{id} + \text{id} \times f \} \\
& \begin{cases} \text{avg} \cdot \text{in} = [\pi_1 \cdot b, \pi_1 \cdot q] \cdot (\text{id} + \text{id} \times \langle \text{avg}, \text{length} \rangle) \\ \text{length} \cdot \text{in} = [\pi_2 \cdot b, \pi_2 \cdot q] \cdot (\text{id} + \text{id} \times \langle \text{avg}, \text{length} \rangle) \end{cases} \\
\equiv & \{ \text{Definição de in para as listas ([nil, cons]) e Lei 22 - Absorção + } \} \\
& \begin{cases} [\text{avg} \cdot \text{nil}, \text{avg} \cdot \text{cons}] = [\pi_1 \cdot b \cdot \text{id}, \pi_1 \cdot q \cdot (\text{id} \times \langle \text{avg}, \text{length} \rangle)] \\ [\text{length} \cdot \text{nil}, \text{length} \cdot \text{cons}] = [\pi_2 \cdot b \cdot \text{id}, \pi_2 \cdot q \cdot (\text{id} \times \langle \text{avg}, \text{length} \rangle)] \end{cases} \\
\equiv & \{ \text{Lei 27 - Eq +, 2 vezes; Lei 1, Natural-id } \} \\
& \begin{cases} \text{avg} \cdot \text{nil} = \pi_1 \cdot b \\ \text{avg} \cdot \text{cons} = \pi_1 \cdot q \cdot (\text{id} \times \langle \text{avg}, \text{length} \rangle) \\ \text{length} \cdot \text{nil} = \pi_2 \cdot b \\ \text{length} \cdot \text{cons} = \pi_2 \cdot q \cdot (\text{id} \times \langle \text{avg}, \text{length} \rangle) \end{cases} \\
\equiv & \{ \text{Em Haskell, avg [] = 0 e length [] = 0 } \} \\
& \begin{cases} \pi_1 \cdot b = 0 \\ \text{avg} \cdot \text{cons} = \pi_1 \cdot q \cdot (\text{id} \times \langle \text{avg}, \text{length} \rangle) \\ \pi_2 \cdot b = 0 \\ \text{length} \cdot \text{cons} = \pi_2 \cdot q \cdot (\text{id} \times \langle \text{avg}, \text{length} \rangle) \end{cases} \\
\equiv & \{ \text{Lei 71 - Igualdade Extensional; Lei 72 - Def-Comp } \} \\
& \begin{cases} \pi_1 (b \ l) = 0 \\ \text{avg} (h : t) = (\pi_1 (q ((\text{id} \times \langle \text{avg}, \text{length} \rangle) (h : t)))) \\ \pi_2 (b \ l) = 0 \\ \text{length} (h : t) = (\pi_2 (q ((\text{id} \times \langle \text{avg}, \text{length} \rangle) (h : t)))) \end{cases} \\
\equiv & \{ \text{Lei 77 - Def-x; Lei 73 - Def-id } \} \\
& \begin{cases} \pi_1 (b \ l) = 0 \\ \text{avg} (h : t) = (\pi_1 (q (h, \langle \text{avg}, \text{length} \rangle (t)))) \\ \pi_2 (b \ l) = 0 \\ \text{length} (h : t) = (\pi_2 (q (h, \langle \text{avg}, \text{length} \rangle (t)))) \end{cases} \\
\equiv & \{ \text{Lei 76 - Def-split } \}
\end{aligned}$$

$$\begin{cases} \pi_1 (b \ l) = 0 \\ avg (h : t) = (\pi_1 (q (h, (avg \ t, length \ t)))) \\ \pi_2 (b \ l) = 0 \\ length (h : t) = (\pi_2 (q (h, (avg \ t, length \ t)))) \end{cases}$$

□

$$avg = \pi_1 \cdot avg_aux$$

Como dito no enunciado, o gene será um $[b, q]$. Através da nossa demonstração, percebemos que o lado esquerdo do either (ou seja, b) será um either, em que as suas alternativas serão funções constantes de 0 ($\pi_1 (b \ l) = 0$ e $\pi_2 (b \ l) = 0$).

Já o seu lado direito, como podemos ver na demonstração, será apenas uma função que realiza tanto a média como o comprimento de uma lista através do comprimento e da média do resto da lista, guardando no primeiro elemento de um par a média e no segundo o comprimento.

$$avg_aux = cataList [(0, 0), aux] \textbf{ where}$$

$$aux (x, (md, c)) = ((x + md * c) / (c + 1), c + 1)$$

Solução para árvores de tipo **LTree**:

$$\begin{aligned} avg_aux &= \llbracket [b, q] \rrbracket \\ &\equiv \{ \text{Definição de } avg_aux \} \\ \langle avg, length \rangle &= \llbracket [b, q] \rrbracket \\ &\equiv \{ \text{Resultado calculado em cima} \} \\ \langle avg, length \rangle &= \llbracket \langle [\pi_1 \cdot b, \pi_1 \cdot q], [\pi_2 \cdot b, \pi_2 \cdot q] \rangle \rrbracket \\ &\equiv \{ \text{Lei 52 - Fokkinga e Functor de LTree: } F f = id + f^2 \} \\ \begin{cases} avg \cdot \mathbf{in} = [\pi_1 \cdot b, \pi_1 \cdot q] \cdot (id + \langle avg, length \rangle \uparrow 2) \\ length \cdot \mathbf{in} = [\pi_2 \cdot b, \pi_2 \cdot q] \cdot (id + \langle avg, length \rangle \uparrow 2) \end{cases} \\ &\equiv \{ \text{Definição de in para as LTree ([Leaf, Fork]) e Lei 22 - Absorção +} \} \\ \begin{cases} [avg \cdot Leaf, avg \cdot Fork] = [\pi_1 \cdot b \cdot id, \pi_1 \cdot q \cdot (\langle avg, length \rangle \uparrow 2)] \\ [length \cdot Leaf, length \cdot Fork] = [\pi_2 \cdot b \cdot id, \pi_2 \cdot q \cdot (\langle avg, length \rangle \uparrow 2)] \end{cases} \\ &\equiv \{ \text{Lei 27 - Eq +, 2 vezes; Lei 1, Natural-id} \} \\ \begin{cases} avg \cdot Leaf = \pi_1 \cdot b \\ avg \cdot Fork = \pi_1 \cdot q \cdot (\langle avg, length \rangle \uparrow 2) \\ length \cdot Leaf = \pi_2 \cdot b \\ length \cdot Fork = \pi_2 \cdot q \cdot (\langle avg, length \rangle \uparrow 2) \end{cases} \\ &\equiv \{ \text{Lei 71 - Igualdade Extensional} \} \\ \begin{cases} avg (Leaf \ lf) = \pi_1 (b \ lf) \\ avg (Fork \ (fl, fr)) = \pi_1 (q (\langle avg, length \rangle \uparrow 2)) (fl, fr) \\ length (Leaf \ lf) = \pi_2 (b \ lf) \\ length (Fork \ (fl, fr)) = \pi_2 (q (\langle avg, length \rangle \uparrow 2)) (fl, fr) \end{cases} \\ &\equiv \{ \text{Propriedade do quadrado de um número} \} \\ \begin{cases} avg (Leaf \ lf) = \pi_1 (b \ lf) \\ avg (Fork \ (fl, fr)) = \pi_1 (q (\langle avg, length \rangle \times \langle avg, length \rangle)) (fl, fr) \\ length (Leaf \ lf) = \pi_2 (b \ lf) \\ length (Fork \ (fl, fr)) = \pi_2 (q (\langle avg, length \rangle \times \langle avg, length \rangle)) (fl, fr) \end{cases} \\ &\equiv \{ \text{Lei 77 - Def-x} \} \\ \begin{cases} avg (Leaf \ lf) = \pi_1 (b \ lf) \\ avg (Fork \ (fl, fr)) = \pi_1 (q (\langle avg, length \rangle (fl, fr), \langle avg, length \rangle (fl, fr))) \\ length (Leaf \ lf) = \pi_2 (b \ lf) \\ length (Fork \ (fl, fr)) = \pi_2 (q (\langle avg, length \rangle (fl, fr), \langle avg, length \rangle (fl, fr))) \end{cases} \end{aligned}$$

$$\begin{aligned}
&\equiv \{ \text{Lei 76 - Def-split} \} \\
&\quad \left\{ \begin{array}{l} \text{avg} (\text{Leaf } lf) = \pi_1 (b \text{ } lf) \\ \text{avg} (\text{Fork } (fl, fr)) = \pi_1 (q ((\text{avg } fl, \text{length } fr), (\text{avg } fl, \text{length } fr))) \\ \text{length} (\text{Leaf } lf) = \pi_2 (b \text{ } lf) \\ \text{length} (\text{Fork } (fl, fr)) = \pi_2 (q ((\text{avg } fl, \text{length } fr), (\text{avg } fl, \text{length } fr))) \end{array} \right. \\
&\equiv \{ \text{avg} (\text{Leaf } lf) = lf ; \text{length} (\text{Leaf } lf) = 1 \} \\
&\quad \left\{ \begin{array}{l} lf = \pi_1 (b \text{ } lf) \\ \text{avg} (\text{Fork } (fl, fr)) = \pi_1 (q ((\text{avg } fl, \text{length } fr), (\text{avg } fl, \text{length } fr))) \\ 1 = \pi_2 (b \text{ } lf) \\ \text{length} (\text{Fork } (fl, fr)) = \pi_2 (q ((\text{avg } fl, \text{length } fr), (\text{avg } fl, \text{length } fr))) \end{array} \right. \\
&\square
\end{aligned}$$

Como dito no enunciado, o gene será um $[b, q]$. Através da nossa demonstração, percebemos que o lado esquerdo do either (ou seja, b) será um par, onde o lado esquerdo conservar-se-á um valor de uma Leaf, e no lado direito de b será o valor 1.

Já o seu lado direito do either, como podemos ver na demonstração, será apenas uma função que realiza tanto a média como o comprimento de uma LTree, guardando no primeiro elemento de um par a média dos valores e no segundo o comprimento dos 2 ramos da árvore.

$$\begin{aligned}
\text{avgLTree} &= \pi_1 \cdot \langle \text{gene} \rangle \text{ where} \\
\text{gene} &= [\lambda l \rightarrow (l, 1), \text{aux}] \\
\text{aux} ((md1, c1), (md2, c2)) &= ((md1 * c1 + c2 * md2) / (c1 + c2), c1 + c2)
\end{aligned}$$

Problema 5

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

Index

- LaTeX, [1](#)
 - `bibtex`, [2](#)
 - `lhs2TeX`, [1](#)
 - `makeindex`, [2](#)
- Combinador “pointfree”
 - cata*, [8](#), [9](#), [17](#), [18](#)
 - either*, [3](#), [8](#), [13–19](#)
- Curvas de Bézier, [6](#), [7](#)
- Cálculo de Programas, [1](#), [2](#), [5](#)
 - Material Pedagógico, [1](#)
 - BTree.hs, [8](#)
 - Cp.hs, [8](#)
 - LTree.hs, [8](#), [18](#)
 - Nat.hs, [8](#)
- Deep Learning), [3](#)
- DSL (linguagem específica para domínio), [3](#)
- F#, [8](#), [19](#)
- Functor, [5](#), [11](#)
- Função
 - π_1 , [6](#), [9](#), [17–19](#)
 - π_2 , [9](#), [13](#), [17–19](#)
 - for*, [6](#), [9](#), [15](#)
 - length*, [8](#), [17–19](#)
 - map*, [11](#), [12](#)
 - uncurry*, [3](#)
- Haskell, [1](#), [2](#), [8](#)
 - Gloss, [2](#), [11](#)
 - interpretador
 - GHCi, [2](#)
 - Literate Haskell, [1](#)
 - QuickCheck, [2](#)
 - Stack, [2](#)
- Números de Catalan, [6](#), [10](#)
- Números naturais (\mathbb{N}), [5](#), [6](#), [9](#)
- Programação
 - dinâmica, [5](#)
 - literária, [1](#)
- Racionais, [7](#), [8](#), [10–12](#)
- U.Minho
 - Departamento de Informática, [1](#)

References

- [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.