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

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	16
a93322	Tiago Costa
a93227	Pedro Paulo Tavares
a93221	André Vaz

1 Preâmbulo

Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação (conjunto de leis universais e seus corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos dois cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em Haskell (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas, a validá-los e a produzir textos técnico-científicos de qualidade.

2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *Um programa e a sua documentação devem coincidir.*

Por outras palavras, o código fonte e a documentação de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp2021t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2021t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2021t.zip` e executando:

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

em que `lhs2tex` é um pre-processador que faz "pretty printing" de código Haskell em LaTeX e que deve desde já instalar executando

```
$ cabal install lhs2tex --lib
```

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

```
$ ghci cp2021t.lhs
```

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

Abra o ficheiro `cp2021t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

```
\begin{code}
...
\end{code}
```

é seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de 3 (ou 4) alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na página da disciplina na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo D com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com BibTEX) e o índice remissivo (com `makeindex`),

```
$ bibtex cp2021t.aux
$ makeindex cp2021t.idx
```

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

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

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

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

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

```
> quickCheckWith stdArgs { maxSuccess = 200, maxSize = 10 } prop
+++ OK, passed 200 tests.
```

Qualquer programador tem, na vida real, de ler e analisar (muito!) código escrito por outros. No anexo C disponibiliza-se algum código Haskell relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

3.1 Stack

O Stack é um programa útil para criar, gerir e manter projetos em Haskell. Um projeto criado com o Stack possui uma estrutura de pastas muito específica:

- Os módulos auxiliares encontram-se na pasta *src*.
- O módulos principal encontra-se na pasta *app*.
- A lista de depêndencias externas encontra-se no ficheiro *package.yaml*.

Pode aceder ao GHCi utilizando o comando:

```
stack ghci
```

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

Para gerar o PDF, garanta que se encontra na diretoria *app*.

Problema 1

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

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

- *Symbolic differentiation*
- *Automatic differentiation*

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

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

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

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

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

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

O construtor E simboliza o exponencial de base e.

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

$$Bin\ Sum\ X\ (N\ 10)$$

designa $x + 10$ na notação matemática habitual.

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

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

   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, $inExpAr \cdot outExpAr = id$ e $outExpAr \cdot idExpAr = id$:*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

   - Regra da soma:
$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}(f(x)) + \frac{d}{dx}(g(x))$$

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

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

- Regra do produto:

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

Defina o gene do catamorfismo que ocorre na função

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

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

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

$$
\begin{array}{l}
prop\_const\_rule :: (Real\ a, Floating\ a) \Rightarrow a \rightarrow Bool \\
prop\_const\_rule\ a = sd\ (N\ a) \equiv N\ 0 \\
prop\_var\_rule :: Bool \\
prop\_var\_rule = sd\ X \equiv N\ 1 \\
prop\_sum\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow ExpAr\ a \rightarrow Bool \\
prop\_sum\_rule\ exp1\ exp2 = sd\ (Bin\ Sum\ exp1\ exp2) \equiv sum\_rule\ \mathbf{where} \\
\quad sum\_rule = Bin\ Sum\ (sd\ exp1)\ (sd\ exp2) \\
prop\_product\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow ExpAr\ a \rightarrow Bool \\
prop\_product\_rule\ exp1\ exp2 = sd\ (Bin\ Product\ exp1\ exp2) \equiv prod\_rule\ \mathbf{where} \\
\quad prod\_rule = Bin\ Sum\ (Bin\ Product\ exp1\ (sd\ exp2))\ (Bin\ Product\ (sd\ exp1)\ exp2) \\
prop\_e\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow Bool \\
prop\_e\_rule\ exp = sd\ (Un\ E\ exp) \equiv Bin\ Product\ (Un\ E\ exp)\ (sd\ exp) \\
prop\_negate\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \rightarrow Bool \\
prop\_negate\_rule\ exp = sd\ (Un\ Negate\ exp) \equiv Un\ Negate\ (sd\ exp)
\end{array}
$$

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

   Defina o gene do catamorfismo que ocorre na função

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

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

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

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

Problema 2

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

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

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

[4] Lei (3.94) em [2], página 98.

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

Obter-se-á de imediato

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

usando as regras seguintes:

- O corpo do ciclo loop terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.[5]
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável n.
- Em init coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios do segundo grau $ax^2 + bx + c$ em $\mathbb{N}_0$. Seguindo o método estudado nas aulas[6], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

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

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

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

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

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

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

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

que implemente esta função.

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

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

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

Problema 3

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

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

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

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

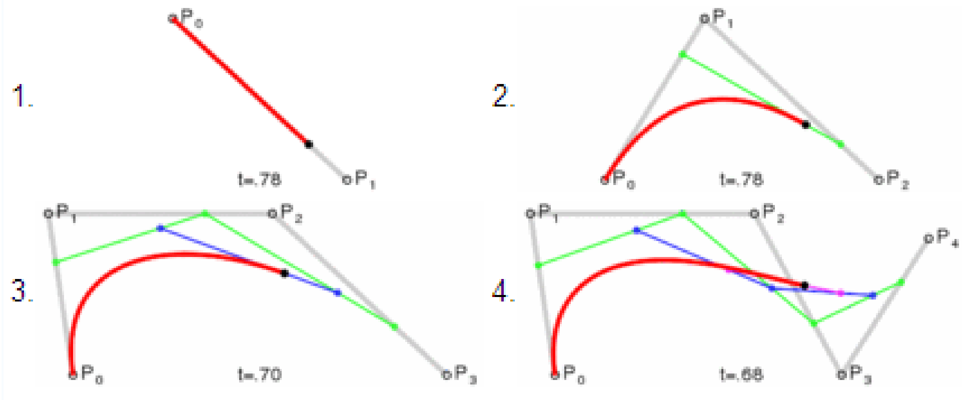

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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

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

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

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

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

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

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

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

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

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

O anexo C tem definida a função

$$\mathit{calcLine} :: \mathit{NPoint} \rightarrow (\mathit{NPoint} \rightarrow \mathit{OverTime}\ \mathit{NPoint})$$

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

$$\mathit{deCasteljau} :: [\mathit{NPoint}] \rightarrow \mathit{OverTime}\ \mathit{NPoint}$$

que implementa o algoritmo respectivo.

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

   Propriedade [QuickCheck] 8 *Definição alternativa.*

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

2. Implemente a função $\mathit{deCasteljau}$ como um hilomorfismo, testando agora a propriedade:

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

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

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

Problema 4

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

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

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

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

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

1. Recorra à lei de recursividade mútua para derivar a função $avg\_aux = (\![[b, q]]\!)$ tal que $avg\_aux = \langle avg, length \rangle$ em listas não vazias.

2. Generalize o raciocínio anterior para o cálculo da média de todos os elementos de uma LTree recorrendo a uma única travessia da árvore (i.e. catamorfismo).

Verifique as suas funções testando a propriedade seguinte:

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

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

Problema 5

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

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

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

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

Anexos

A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

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

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

B Programação dinâmica por recursividade múltipla

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

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

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

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

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

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

[8] Exemplos tirados de [2].
[9] Cf. [2], página 102.

C Código fornecido

Problema 1

$$
\begin{aligned}
&expd :: Floating\ a \Rightarrow a \rightarrow a \\
&expd = Prelude.exp \\
&\textbf{type}\ OutExpAr\ a = () + (a + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a)))
\end{aligned}
$$

Problema 2

Definição da série de Catalan usando factoriais (1):

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

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

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

Problema 3

Algoritmo:

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

Função auxiliar:

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

2D:

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

Modelo:

$$
\begin{aligned}
&\textbf{data}\ World = World\ \{points :: [NPoint] \\
&\quad , time :: Float \\
&\quad \} \\
&initW :: World \\
&initW = World\ [\,]\ 0
\end{aligned}
$$

[10] Fonte: Wikipedia.

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

Gloss:

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

Animação:

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

Propriedades e *main*:

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

Compilação e execução dentro do interpretador:[11]

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

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

QuickCheck

Código para geração de testes:

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

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

Outras funções auxiliares

Lógicas:

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

D Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções para os exercícios propostos, de acordo com o "layout"que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto, disgramas e/ou outras funções auxiliares que sejam necessárias.

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

Problema 1

São dadas:

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

$$
\begin{aligned}
&eval\_exp :: Floating\ a \Rightarrow a \to (ExpAr\ a) \to a\\
&eval\_exp\ a = cataExpAr\ (g\_eval\_exp\ a)\\
&optmize\_eval :: (Floating\ a, Eq\ a) \Rightarrow a \to (ExpAr\ a) \to a\\
&optmize\_eval\ a = hyloExpAr\ (gopt\ a)\ clean\\
&sd :: Floating\ a \Rightarrow ExpAr\ a \to ExpAr\ a\\
&sd = \pi_2 \cdot cataExpAr\ sd\_gen\\
&ad :: Floating\ a \Rightarrow a \to ExpAr\ a \to a\\
&ad\ v = \pi_2 \cdot cataExpAr\ (ad\_gen\ v)
\end{aligned}
$$

Definir:

Para definirmos a função outExpAr temos de ter em atenção o tipo de saída, ou seja, OutExpAr. É então fácil definir esta função tal como todos os outros tipos já previamente definidos. Para o construtor X inserimos o único elemento do tipo 1 na parte esquerda do coproduto. E seguimos o mesmo raciocínio para os outros construtores.

$$
\begin{aligned}
&outExpAr :: ExpAr\ a \to OutExpAr\ a\\
&outExpAr\ X = i_1\ ()\\
&outExpAr\ (N\ a) = i_2 \cdot i_1\ \$\ a\\
&outExpAr\ (Bin\ op\ a\ b) = i_2 \cdot i_2 \cdot i_1\ \$\ (op, (a, b))\\
&outExpAr\ (Un\ op\ a) = i_2 \cdot i_2 \cdot i_2\ \$\ (op, a)
\end{aligned}
$$

A função recExpAr é o functor do tipo ExpAr, ou seja, queremos preservar os respetivos construtores e aplicar a função g à expressão.

$$recExpAr\ g = baseExpAr\ id\ id\ id\ g\ g\ id\ g$$

A função eval_exp é um catamorfismo, logo temos de definir o gene desta. Assim, a função g_eval_exp é o gene deste catamorfismo. Através de um diagrama é fácil de ver o catamorfismo.

$$
\begin{tikzcd}
ExpAr\ A \arrow[d, "(\!|g\_eval\_exp|\!)"'] & & OutExpAr\ A \arrow[ll, "\mathsf{in}"'] \arrow[d, "recExpAr\ eval\_exp"] \\
A & & 1 + (A + ((BinOp, (A, A)) + (UnOp, A))) \arrow[ll, "g\_eval\_exp"]
\end{tikzcd}
$$

Através do diagrama conseguimos perceber que para avaliar uma expressão temos vários pontos :

- Se tivermos X, temos que dar o valor que recebemos.
- Se tivermos uma constante (N *natural*) devolvemos esse natural
- Se tivermos uma operação binária temos de aplicar essa operação de forma *uncurried* porque recebemos um par
- Se tivermos uma operação unária basta aplicar a função

$$
\begin{aligned}
&g\_eval\_exp\ a = [\underline{a}, [id, [\widehat{binOp}, \widehat{unOp}]]]\ \mathbf{where}\\
&\quad binOp\ Sum = \widehat{(+)}\\
&\quad binOp\ Product = \widehat{(*)}\\
&\quad unOp\ Negate = negate\\
&\quad unOp\ E = expd
\end{aligned}
$$

Para a otimização desta avaliação de expressões podemos implementar esta avaliação como um hilomorfismo. A vantagem desta forma de avaliação em vez de um "gene"inteligente é o facto de se a expressão crescer de forma descontrolada e, por vezes, sem necessidade porque é um produto por zero.

Assim, o passo de *divide* deste hilomorfismo, é a funçao *clean*, que tira partido dos elemntos absorventes das operações. Ou seja, o resultado do produto por zero de uma expressão pode ser logo zero.

$$
\begin{aligned}
&clean\ X = i_1\ ()\\
&clean\ (N\ a) = i_2\ (i_1\ a)
\end{aligned}
$$

$$
\begin{array}{l}
clean\ (Bin\ Sum\ (N\ 0)\ a) = clean\ a\\
clean\ (Bin\ Sum\ a\ (N\ 0)) = clean\ a\\
clean\ (Bin\ Product\ (N\ 0)\ \_) = i_2\ (i_1\ 0)\\
clean\ (Bin\ Product\ \_\ (N\ 0)) = i_2\ (i_1\ 0)\\
clean\ (Bin\ op\ a\ b) = i_2 \cdot i_2 \cdot i_1\ \$\ (op, (a, b))\\
clean\ (Un\ Negate\ (Un\ Negate\ a)) = clean\ a\\
clean\ (Un\ op\ a) = i_2 \cdot i_2 \cdot i_2\ \$\ (op, a)
\end{array}
$$

$$
gopt\ a = g\_eval\_exp\ a
$$

Para a derivada de uma expressão seguindo *symbolic differentiation*, alterando a expressão de forma recursiva. Para o gene deste catamorfismo temos de obrigatoriamente dar um par de expressões porque na primeira componente do par temos de preservar a expressão original para a regra do produto. Deste modo temos que o gene deste catamorfismo é composto pelas várias regras da derivação :

- Se recebemos um X então a sua derivada é 1. O par é então (X,N 1)
- Se recebemos um número natural devolvemos o par (N x, N 0)
- Se recebemos a soma de duas expressões já derivadas é apenas darmos a soma dessas derivadas
- Se recebemos o produto então temos de aplicar a regra do produto usando a pré-condição de termos a expressão original reservada
- No caso quer da exponenciação quer da negção é apenas aplicar a regra da exponenciação e negar a derivada da expressão resultante respetivamente

$$
\begin{array}{l}
sd\_gen :: Floating\ a \Rightarrow\\
\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \rightarrow (ExpAr\ a\\
sd\_gen = [sd1, [sd2, [sd3, sd4]]]\ \mathbf{where}\\
\quad sd1\ x = (X, (N\ 1))\\
\quad sd2\ y = ((N\ y), (N\ 0))\\
\quad sd3\ (Sum, ((a, b), (c, d))) = (Bin\ Sum\ a\ c, Bin\ Sum\ b\ d)\\
\quad sd3\ (Product, ((a, b), (c, d))) = (Bin\ Product\ a\ c, Bin\ Sum\ (Bin\ Product\ a\ d)\ (Bin\ Product\ b\ c))\\
\quad sd4\ (E, (a, b)) = (Un\ E\ a, Bin\ Product\ (Un\ E\ a)\ b)\\
\quad sd4\ (Negate, (a, b)) = (Un\ Negate\ a, Un\ Negate\ b)
\end{array}
$$

Como referido a técnica de derivação seguindo *symbolic differentiation* não é a mais efeciente, havendo por isso a possibilidade de derivar a expressão no ponto. Assim, o gene deste catamorfismo torna-se fácil de definir. Usando o mesmo diagrama definido mais acima neste documento basta substituir o tipo de retorno por Naturais e, assim, passamos não a lidar com expressões mas sim com números, sendo a derivação mais fácil.

$$
\begin{array}{l}
ad\_gen\ v = [g1, [g2, [g3, g4]]]\ \mathbf{where}\\
\quad g1\ x = (v, 1)\\
\quad g2\ y = (y, 0)\\
\quad g3\ (Sum, ((a, b), (c, d))) = (a + c, b + d)\\
\quad g3\ (Product, ((a, b), (c, d))) = (a * c, (a * d) + (b * c))\\
\quad g4\ (E, (a, b)) = ((expd\ a), (expd\ a) * b)\\
\quad g4\ (Negate, (a, b)) = ((negate\ a), negate\ b)
\end{array}
$$

Problema 2

Definir

$$
\begin{array}{l}
c\ 0 = 1\\
c\ (n + 1) = ((c\ n) * f\ n)\ `div`\ h\ n\\
f\ 0 = 2\\
f\ (n + 1) = faux\ n + f\ n\\
faux\ 0 = 10
\end{array}
$$

$$
\begin{array}{l}
\mathit{faux}\ (n+1) = 8 + \mathit{faux}\ n \\
h\ 0 = 2 \\
h\ (n+1) = \mathit{haux}\ n + h\ n \\
\mathit{haux}\ 0 = 4 \\
\mathit{haux}\ (n+1) = 2 + \mathit{haux}\ n \\
\mathit{loop}\ (c, f, \mathit{faux}, h, \mathit{haux}) = ((c * f)\ \text{`}\mathit{div}\text{`}\ h, \mathit{faux} + f, 8 + \mathit{faux}, \mathit{haux} + h, 2 + \mathit{haux}) \\
\mathit{inic} = (1, 2, 10, 2, 4) \\
\mathit{prj}\ (c, f, \mathit{faux}, h, \mathit{haux}) = c
\end{array}
$$

por forma a que

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

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

Partindo da expressão dada podemo-la expressar como uma recursão.

$$
\begin{array}{l}
c\ 0 = 1 \\
c\ (n+1) = (c\ n) * (2\ n + 2) * (2\ n + 1) \div (n + 2) * (n + 1)
\end{array}
$$

De notar que decidimos agrupar a chamada de C_n com ((2n +2) * (2n +1)) e só depois realizar a divisão graças a ser a divisão inteira, e , por isso, não realizarmos arredondamentos.

Seguidamente, seguindo a regra de algibeira dada, temos então de transformar estas dependências de n por funções que sejam também elas recursivas e não dependam de n.

Definimos assim as seguintes funções :

$$
\begin{array}{l}
f\ n = (2\ n + 2) * (2\ n + 1) \\
\mathit{faux}\ n = 8\ n + 10 \\
h\ n = (n + 2) * (n + 1) \\
\mathit{haux}\ n = 2\ n + 4
\end{array}
$$

Após isto é apenas aplicar de forma direta a regra dada.

Problema 3

$$
\begin{array}{l}
\mathit{calcLine} :: \mathit{NPoint} \to (\mathit{NPoint} \to \mathit{OverTime}\ \mathit{NPoint}) \\
\mathit{calcLine} = \mathit{cataList}\ h\ \textbf{where} \\
\quad h = [\mathit{h1}, \mathit{h2}] \\
\quad \mathit{h1}\ x = \underline{\mathit{nil}} \\
\quad \mathit{h2}\ (d, f)\ [\,] = \mathit{nil} \\
\quad \mathit{h2}\ (d, f)\ (x : \mathit{xs}) = \lambda z \to \mathit{concat}\ \$\ (\mathit{sequenceA}\ [\mathit{singl} \cdot \mathit{linear1d}\ d\ x, f\ \mathit{xs}])\ z
\end{array}
$$

Para definir o catamorfismo *calcLine* baseamo-nos na função auxiliar dada e transformamo-la num catamorfismo. Assim, temos que o diagrama deste catamorfismo é :

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

Deste modo o gene torna-se fácil de compreender. Caso a lista recebida seja vazia temos dedar uma função que receba 2 argumentos e produza a lista vazia. Caso contrário temos o "passo de recursão"e o que temos de fazer é calcular a tal interpolação linear.

$$
\begin{array}{l}
\mathit{deCasteljau}\ [\,] = \mathit{nil} \\
\mathit{deCasteljau}\ l = (\mathit{hyloAlgForm}\ \mathit{alg}\ \mathit{coalg})\ l\ \textbf{where} \\
\quad \mathit{coalg} = (\mathit{id} + \langle \mathit{cons} \cdot (\mathit{id} \times \mathit{init}), \pi_2 \rangle) \cdot \mathit{outNVL}
\end{array}
$$

$$
\begin{aligned}
&\quad alg = [alg1, alg2] \ \mathbf{where} \\
&\qquad alg1 \ x = \underline{x} \\
&\qquad alg2 \ (f, g) = \lambda pt \rightarrow calcLine \ (f \ pt) \ (g \ pt) \ pt \\
&hyloAlgForm = hyloLTree
\end{aligned}
$$

Para definirmos o hylomorfismo deste problema baseamo-nos na função auxiliar dada pelo professor.Deste modo inferimos que a estrutura auxiliar que o anamorfismo deveria criar era um LTree em que cada folha (Leaf) tem elementos do tipo NPoint. Assim, aquando da aplicação do catamorfismo, basta aplicar ao Fork a função calcLine que obtemos o tipo OverTime NPoint.Partimos também do principio que a lista que recebemos não é vazia porque fazemos primeiro essa verificação e , por isso, usamos a definiçao de outNVL para o anamorfismo. Diagrama que fundamenta melhor o esquema do hylomorfismo :

Anamorfismo :

$$
\begin{tikzcd}
{[NPoint]} \arrow[r, "(id+\langle cons\cdot(id\times init),\pi_2\rangle)\cdot outNVL"] \arrow[d, "coalg"] & NPoint + [NPoint] * [NPoint] \arrow[d, "id+coalg*coalg"] \\
\mathsf{LTree}\ NPoint & NPoint + \mathsf{LTree}\ NPoint * \mathsf{LTree}\ NPoint \arrow[l, "inLTree"]
\end{tikzcd}
$$

Catamorfismo :

$$
\begin{tikzcd}
\mathsf{LTree}\ NPoint \arrow[d, "alg"'] & NPoint + \mathsf{LTree}\ NPoint * \mathsf{LTree}\ NPoint \arrow[l, "\mathbf{in}"'] \arrow[d, "id+alg*alg"] \\
OverTime\ NPoint & NPoint + OverTime\ NPoint * OverTime\ NPoint \arrow[l, "{[alg1, alg2]}"]
\end{tikzcd}
$$

Problema 4

Solução para listas não vazias:

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

Para este problema é necessário definir o "conjunto de leis"dos catamorfismos para listas não vazia.Assim,definimos o *inNVl*, o *outNVL*, o *recNVL* e claro, por fim, o *cataNVL* . Depois de termos este conjunto definido temos de olhar para o gene deste catamorfismo e perceber 2 coisas :

A função length pode ser calculada de forma pointwise seguindo o diagrama.

$$
\begin{tikzcd}
A* \arrow[d, "<avg,length>"'] & A + A * (A*) \arrow[l, "inNVL"'] \arrow[d, "id+id*<avg,length>"] \\
A & A + A * (A * A) \arrow[l, "len\_gen"]
\end{tikzcd}
$$

A partir deste podemos facilmente obter o gene (len_gen):

$$
\begin{aligned}
&len\_gen = [g1, g2] \ \mathbf{where} \\
&g1 \ x = [x] \\
&g2 \ (x, (avg, len)) = 1 + len
\end{aligned}
$$

Mas a partir das leis do cálculo podemos tornar então em pointfree

$$
\begin{aligned}
& \quad g1 \ x = [x] \\
\equiv & \qquad \{ \text{ def. singl } \} \\
& \quad g1 \ x = singl \ x \\
\equiv & \qquad \{ \text{ igualdade extensional } \} \\
& \quad g1 = singl
\end{aligned}
$$

$\square$

Da mesma maneira podemos então a partir de g2 derivar a sua versão pointfree

$$
\begin{array}{ll}
& g2\ (x, (avg, len)) = 1 + len \\
\equiv & \quad \{ \text{ def. succ } \} \\
& g2\ (x, (avg, len)) = \mathsf{succ}\ \ len \\
\equiv & \quad \{ \text{ def.comp, def.proj } \} \\
& g2\ (x, (avg, len)) = \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\ (x, (avg, len)) \\
\equiv & \quad \{ \text{ igualdade extensional } \} \\
& g2 = \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \\
\Box &
\end{array}
$$

Seguindo o mesmo raciocínio e usando o mesmo diagrama, obtemos a *avg*.
Neste momento temos então um split de eithers mas queremos um either de splits.Momento oportuno para aplicarmos a Lei da Troca e obtermos o que abaixo está definido :

$$
\begin{array}{l}
inNVL = [singl, cons] \\
outNVL\ [x] = i_1\ x \\
outNVL\ (a : l) = i_2\ (a, l) \\
recNVL\ f = id + id \times f \\
cataNVL\ g = g \cdot recNVL\ (cataNVL\ g) \cdot outNVL \\
avg\_aux = cataNVL\ [\langle id, \underline{1} \rangle, \langle algorit, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle]\ \mathbf{where} \\
\quad algorit\ (x, (avg, len)) = (x + len * avg)\ /\ (1 + len)
\end{array}
$$

Solução para árvores de tipo LTree:

$$
\begin{array}{l}
avgLTree = \pi_1 \cdot (\!|\ gene\ |\!)\ \mathbf{where} \\
\quad gene = [\langle id, \underline{1} \rangle, \langle avgLTreeaux, \widehat{(+)} \cdot (\pi_2 \times \pi_2) \rangle]\ \mathbf{where} \\
\quad\quad avgLTreeaux\ ((avgLeft, lenLeft), (avgRight, lenRight)) = \\
\quad\quad\quad ((avgLeft * lenLeft) + (avgRight * lenRight))\ /\ (lenLeft + lenRight)
\end{array}
$$

Para o tipo LTree podemos também, mais uma vez, aproveitar o diagrama para construirmos o gene deste catamorfismo.

$$
\begin{tikzcd}
\mathsf{LTree}\ A \arrow[d, "\langle avgLTree, lenLTree \rangle"'] & A + (\mathsf{LTree}\ A * \mathsf{LTree}\ A) \arrow[l, "inLTree"'] \arrow[d, "id + (\langle avgLTree, lenLTree \rangle, \langle avgLTree, lenLTree \rangle)"] \\
A * A & A + ((A * A) * (A * A)) \arrow[l, "tree\_gen"]
\end{tikzcd}
$$

Obtemos então o seguinte gene para a len desta àrvore:

$$
\begin{array}{l}
lenLTree\ (Leaf) = 1 \\
lenLTree\ (Fork\ (e, d)) = lenLTree\ e + lenLTree\ d
\end{array}
$$

Usando as leis do cálculo obtemos então :

$$
\begin{array}{ll}
& \left\{ \begin{array}{l} lenLTree\ (Leaf) = 1 \\ lenLTree\ (Fork\ (e, d)) = lenLTree\ e + lenLTree\ d \end{array} \right. \\
\equiv & \quad \{ \text{ def.comp, def-const, def-x } \} \\
& \left\{ \begin{array}{l} (lenLTree \cdot Leaf)\ x = \underline{1}\ x \\ (lenLTree \cdot Fork)\ (e, d) = \widehat{(+)} \cdot (lenLTree * lenLTree)\ (e, d) \end{array} \right. \\
\equiv & \quad \{ \text{ def.split, def.proj,igualdade extensional } \}
\end{array}
$$

$$\left\{\begin{array}{l}(lenLTree \cdot Leaf) = \underline{1}\\(lenLTree \cdot Fork) = \widehat{(+)} \cdot (\pi_2 \cdot \langle avgLTree, lenLTree\rangle, \pi_2 \cdot \langle avgLTree, lenLTree\rangle)\end{array}\right.$$

$\equiv$ { def. inLtree, def.funtor-x, eq-+,def-fusao }

$$lenLTree \cdot inLtree = [\underline{1}, \widehat{(+)} \cdot (\pi_2 * \pi_2) \cdot (\langle avgLTree, lenLTree\rangle, \langle avgLTree, lenLTree\rangle)]$$

$\equiv$ { def.absorçao-+,def-funtor,universal-cata }

$$len = (\!|\, \underline{1}, (\widehat{(+)} \cdot (\pi_2 * \pi_2))\,|\!)$$

$\square$

Para o gene do avgLTree decidimos deixar em versão pointwise porque é de mais fácil leitura e é a aplicação direta do algoritmo dado. Para finalizar, como na definição anterior, temos um split de eithers e queremos então um either de splits,usamos por isso a Lei da troca e obtemos a definição que nós propusemos.

Problema 5

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

```
open Cp

// (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 -> i1 ()
    | Node (y,(t1,t2)) -> i2 (y,(t1,t2))


// (2) Ana + cata + hylo ---------------------------------------------------
// recBTree g = id -|- (id >< (g >< g))

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

let recBTree g x = baseBTree id g x
let rec cataBTree a x = (a << (recBTree (cataBTree a)) << outBTree) x

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

let hyloBTree a c x = (cataBTree a << anaBTree c) x

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

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

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


let inordt x = cataBTree (inord) x

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

let preordt x = cataBTree preord x

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

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

let rec part p li =
       match li with
       |[] -> ([],[])
       |(h::t) -> if not(p h) then let (s,l) = part p t in (h::s,l) else let (s,l) =


let qsep li =
    match li with
    | [] -> i1 ()
    | (h::t) -> i2(h,(part ((<) h) t) )

let qSort x = hyloBTree inord qsep x

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

let tunion(a,(l,r)) = (List.map (List.append [a]) l)@(List.map(List.append [a]) r)

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

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

let present x = inord x

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

let hanoi x = hyloBTree present strategy x

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


let baldepth x =
        let f((b1,d1),(b2,d2)) = ((b1,b2),(d1,d2))
        let h(a,((b1,b2),(d1,d2))) = (b1 && b2 && abs(d1-d2)<=1,1+max d1 d2)
```

```
        let g x = either (konst(true,1)) (h<<(id><f)) x
        in cataBTree g x

let depthBTree x = p2(baldepth x)
let balBTree x = p1(baldepth x)
```

Índice

Referências

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

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