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

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	7
a93276	Ana Rita Teixeira
a93230	Beatriz da Silva Rodrigues
a93206	Vasco Oliveira Matos

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" [**?**], 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 **??** 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 **??** disponibiliza-se algum código Haskell relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

3.1 Stack

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

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

Pode aceder ao GHCi utilizando o comando:

```
stack ghci
```

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

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

Problema 1

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

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

- *Symbolic differentiation*
- *Automatic differentiation*

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

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

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

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

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

$$
\begin{aligned}
&\textbf{data}\ BinOp = Sum \\
&\quad \mid Product \\
&\quad \textbf{deriving}\ (Eq, Show) \\
&\textbf{data}\ UnOp = Negate \\
&\quad \mid E \\
&\quad \textbf{deriving}\ (Eq, Show)
\end{aligned}
$$

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:

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

   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 $outExpAr \cdot idExpAr = id$:*

$$
\begin{aligned}
&prop\_in\_out\_idExpAr :: (Eq\ a) \Rightarrow ExpAr\ a \to Bool \\
&prop\_in\_out\_idExpAr = inExpAr \cdot outExpAr \equiv id \\
&prop\_out\_in\_idExpAr :: (Eq\ a) \Rightarrow OutExpAr\ a \to Bool \\
&prop\_out\_in\_idExpAr = outExpAr \cdot inExpAr \equiv id
\end{aligned}
$$

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

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

calcula o resultado da expressão. Na página **??** 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.*

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

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

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

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

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

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

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

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

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:

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

[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

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

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.*

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

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

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

   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.*

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

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

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

[4] Lei (3.94) em [?], página 98.

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

Obter-se-á de imediato

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

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* coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

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

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

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

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

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

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

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

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

que implemente esta função.

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

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

Sugestão: Começar por estudar muito bem o processo de cálculo dado no anexo **??** 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 [**?**] e tópico Recursividade mútua nos vídeos das aulas teóricas.

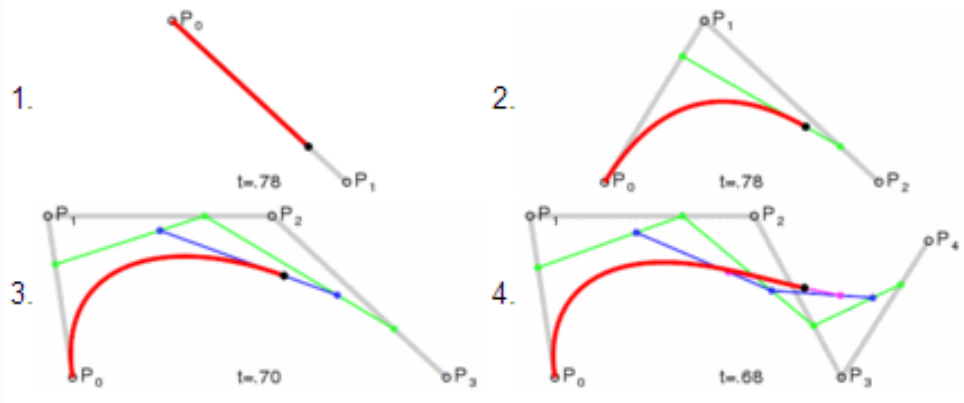


Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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:

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

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

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

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

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

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

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

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

O anexo **??** tem definida a função

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

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

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

que implementa o algoritmo respectivo.

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

   Propriedade [QuickCheck] 8 *Definição alternativa.*

   $$
   \begin{aligned}
   &prop\_calcLine\_def :: NPoint \to NPoint \to Float \to Bool \\
   &prop\_calcLine\_def\ p\ q\ d = calcLine\ p\ q\ d \equiv zipWithM\ linear1d\ p\ q\ d
   \end{aligned}
   $$

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

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

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

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

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

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 $avg\_aux = \langle avg, length\rangle$ 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:*

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

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 **??**. 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]

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

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

$$
\begin{array}{ccc}
\mathbb{N}_0 & \xleftarrow{\ \mathsf{in}\ } & 1 + \mathbb{N}_0 \\
{\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|g|\!)} \\
B & \xleftarrow[\ g\ ]{} & 1 + B
\end{array}
$$

B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da 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 **??** 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 [**?**].
[9] Cf. [**?**], página 102.

C Código fornecido

Problema 1

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

Problema 2

Definição da série de Catalan usando factoriais (**??**):

$$
\begin{spec}
catdef\ n = (2 * n)! \div ((n+1)! * n!)
\end{spec}
$$

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

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

Problema 3

Algoritmo:

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

Função auxiliar:

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

2D:

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

Modelo:

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

[10] Fonte: Wikipedia.

```
tick :: Float → World → World
tick dt world = world {time = (time world) + dt}

actions :: Event → World → World
actions (EventKey (MouseButton LeftButton) Down _ p) world =
  world {points = (points world) ++ [(λ(x, y) → map to_ℚ [x, y]) p]}
actions (EventKey (SpecialKey KeyDelete) Down _ _) world =
  world {points = cond (≡ []) id init (points world)}
actions _ world = world

scaleTime :: World → Float
scaleTime w = (1 + cos (time w)) / 2

bezier2dAtTime :: World → (Float, Float)
bezier2dAtTime w = (bezier2dAt w) (scaleTime w)

bezier2dAt :: World → OverTime (Float, Float)
bezier2dAt w = bezier2d (points w)

thicCirc :: Picture
thicCirc = ThickCircle 4 10

ps :: [Float]
ps = map from_ℚ ps' where
  ps' :: [ℚ]
  ps' = [0, 0.01 .. 1]   -- interval
```

Gloss:

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

Animação:

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

Propriedades e *main*:

```
runBezier :: IO ()
runBezier = play (InWindow "Bézier" (600, 600) (0, 0))
  black 50 initW picture actions tick

runBezierSym :: IO ()
runBezierSym = quickCheckWith (stdArgs {maxSize = 20, maxSuccess = 200}) prop_bezier_sym
```

Compilação e execução dentro do interpretador:[11]

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

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

QuickCheck

Código para geração de testes:

```
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 ?=
(?=) :: Real a ⇒ a → a → Bool
(?=) x y = (to_ℚ x) ≡ (to_ℚ y)
```

Outras funções auxiliares

Lógicas:

```
infixr 0 ⇒
(⇒) :: (Testable prop) ⇒ (a → Bool) → (a → prop) → a → Property
p ⇒ f = λa → p a ⇒ f a
infixr 0 ⇔
(⇔) :: (a → Bool) → (a → Bool) → a → Property
p ⇔ f = λa → (p a ⇒ property (f a)) .&&. (f a ⇒ property (p a))
infixr 4 ≡
(≡) :: Eq b ⇒ (a → b) → (a → b) → (a → Bool)
f ≡ g = λa → f a ≡ g a
infixr 4 ⩽
(⩽) :: Ord b ⇒ (a → b) → (a → b) → (a → Bool)
f ⩽ g = λa → f a ⩽ g a
infixr 4 ∧
(∧) :: (a → Bool) → (a → Bool) → (a → Bool)
f ∧ g = λa → ((f a) ∧ (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
```

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

Definir:

$$
\begin{aligned}
&outExpAr\ X = i_1\ ()\\
&outExpAr\ (N\ a) = i_2\ (i_1\ a)\\
&outExpAr\ (Bin\ op\ x\ y) = i_2\ (i_2\ (i_1\ (op, (x, y))))\\
&outExpAr\ (Un\ op\ x) = i_2\ (i_2\ (i_2\ (op, x)))\\
&\quad --\\
&recExpAr\ g = baseExpAr\ id\ id\ id\ g\ g\ id\ g\\
&\quad --\\
&g\_eval\_exp\ a = [f, g]\\
&\quad \textbf{where}\ f = \underline{a}\\
&\qquad g = [h, j]\\
&\qquad h\ (b) = b\\
&\qquad j = [k, l]\\
&\qquad k\ (Sum, (z, w)) = z + w\\
&\qquad k\ (Product, (z, w)) = z * w\\
&\qquad l\ (Negate, z) = -z\\
&\qquad l\ (E, z) = expd\ z\\
&\quad --\\
&clean\ (Bin\ Sum\ x\ y) \mid x \equiv (N\ 0) = outExpAr\ (y)\\
&\quad \mid y \equiv (N\ 0) = outExpAr\ (x)\\
&\quad \mid otherwise = outExpAr\ (Bin\ Sum\ x\ y)\\
&clean\ (Bin\ Product\ x\ y) \mid x \not\equiv (N\ 0) \wedge y \not\equiv (N\ 0) = outExpAr\ (Bin\ Product\ x\ y)\\
&\quad \mid otherwise = outExpAr\ (N\ 0)\\
&clean\ (Un\ Negate\ x) \mid x \not\equiv (N\ 0) = outExpAr\ (Un\ Negate\ x)\\
&\quad \mid otherwise = outExpAr\ (N\ 0)\\
&clean\ (Un\ E\ x) \mid x \not\equiv (N\ 0) = outExpAr\ (Un\ E\ x)\\
&\quad \mid otherwise = outExpAr\ (N\ 1)\\
&clean\ exp = outExpAr\ exp\\
&\quad --\\
&gopt\ a = [f, g]\\
&\quad \textbf{where}\ f = \underline{a}\\
&\qquad g = [h, j]\\
&\qquad h\ (b) = b\\
&\qquad j = [k, l]\\
&\qquad k\ (Sum, (z, w)) \mid z \equiv 0 \wedge w \equiv 0 = 0\\
&\qquad\quad \mid otherwise = z + w\\
&\qquad k\ (Product, (z, w)) = z * w\\
&\qquad l\ (Negate, z) = -z\\
&\qquad l\ (E, z) \mid z \equiv 0 = 1\\
&\qquad\quad \mid otherwise = expd\ z
\end{aligned}
$$

Para determinar o valor de outExpAr foi resolvido o seguinte sistema de equações:

$$
\begin{aligned}
&out \cdot \mathbf{in} = id\\
\equiv\ &\quad \{\ \text{in definition, fusion-+, universal property ,}\ \}\\
&\begin{cases} out \cdot X = i_1 \\ out \cdot [N, ops] = i_2 \end{cases}\\
\equiv\ &\quad \{\ \text{fusion-+ , universal property, ops definition}\ \}
\end{aligned}
$$

$$
\begin{cases}
out \cdot X = i_1 \\
out \cdot N = i_2 \cdot i_1 \\
out \cdot [bin, Uncurry\ Un] = i_2 \cdot i_2
\end{cases}
$$

$\equiv$ { fusion-+ , universal property }

$$
\begin{cases}
out \cdot X = i_1 \\
out \cdot N = i_2 \cdot i_1 \\
out \cdot bin = i_2 \cdot i_2 \cdot i_1 \\
out \cdot Uncurry\ Un = i_2 \cdot i_2 \cdot i_2
\end{cases}
$$

$\equiv$ { extensional equality , def-comp }

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

$\equiv$ { def op , def uncurry }

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

□

Para determinar o functor, verificou-se o tipo resultante de out e aplicou-se a função às ocorrências de ExpAr .

$$
\begin{array}{ccc}
ExpAr\ a & \xleftarrow{inExpAr} & 1 + (A + (B\ X\ (ExpAr\ a\ X\ ExpAr\ a) + C\ X\ ExpAr\ a)) \\
{\scriptstyle cataExpAr\ gene}\downarrow & & \downarrow{\scriptstyle recExpAr\ (cataExpAr\ gene)} \\
B & \xleftarrow[gene]{} & 1 + (A + (B\ X\ (E\ X\ E) + C\ X\ E))
\end{array}
$$

Na realização da alínea 2 foi essencial o diagrama do catamorfismo de forma a descobrir o tipo do gene e assim, defini-lo.

$$
\begin{array}{ccc}
Expr\ a & \xleftarrow{inExpAr} & 1 + (A + (B\ X\ (Expr\ a\ X\ (Expr\ a) + C\ X\ Expr\ a)) \\
{\scriptstyle cataExpAr\ g\_eval\_exp}\downarrow & & \downarrow{\scriptstyle recExpAr\ (cataExpAr\ g\_eval\_exp)} \\
\mathbb{N}_0 & \xleftarrow[g\_eval\_exp]{} & 1 + (A + (B\ X\ (\mathbb{N}_0, \mathbb{N}_0) + C\ X\ \mathbb{N}_0))
\end{array}
$$

Para a alínea 3 , criou-se um anamorfismo que limpa os casos especiais da soma, do produto, da negação e da exponenciação. O catamorfismo é muito semelhante ao *gEvalExp*, mas também tem em conta os ditos casos especiais.

$$
\begin{array}{ccc}
ExpAr\ a & \xrightarrow[clean]{} & 1 + (A + (B\ X\ (ExpAr\ a\ X\ ExpAr\ a) + C\ X\ ExpAr\ a)) \\
{\scriptstyle anaExpAr\ clean}\downarrow & & \downarrow{\scriptstyle recExpAr\ (anaExpAr\ clean)} \\
ExpAr\ a & \xleftarrow[inExpAr]{} & 1 + (A + (B\ X\ (ExpAr\ a\ X\ ExpAr\ a) + C\ X\ ExpAr\ a)) \\
{\scriptstyle cataExpAr\ gopt}\downarrow & & \downarrow{\scriptstyle recExpAr\ (cataExpAr\ gopt)} \\
\mathbb{N}_0 & \xleftarrow[gopt]{} & 1 + (A + (B\ X\ (\mathbb{N}_0\ X\ \mathbb{N}_0) + C\ X\ \mathbb{N}_0))
\end{array}
$$

Relativamente à alínea 4, para o cálculo da derivada era essencial mantermos acessível o valor da expressão sem derivaçao para além do valor derivado porque, por exemplo, derivar um produto exige

conhecê-la. Assim o gene do catamorfismo devolve sempre um par no qual o primeiro elemento é a expressão e o segundo elemento é a mesma expressão mas derivada de acordo com as regras comuns da derivação.

$$
\begin{aligned}
&sd\_gen :: Floating\ a \Rightarrow \\
&\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \\
&\quad \rightarrow (ExpAr\ a, ExpAr\ a) \\
&sd\_gen = [f, g] \\
&\quad \textbf{where } f = \underline{(X, N\ 1)} \\
&\qquad g = [h, j] \\
&\qquad h\ (a) = (N\ a, N\ 0) \\
&\qquad j = [k, l] \\
&\qquad k\ (Sum, ((z, w), (x, y))) = (Bin\ Sum\ z\ x, Bin\ Sum\ w\ y) \\
&\qquad k\ (Product, ((z, w), (x, y))) = (Bin\ Product\ z\ x, Bin\ Sum\ (Bin\ Product\ z\ y)\ (Bin\ Product\ w\ x)) \\
&\qquad l\ (Negate, (z, w)) = (Un\ Negate\ z, Un\ Negate\ w) \\
&\qquad l\ (E, (z, w)) = (Un\ E\ z, Bin\ Product\ (Un\ E\ z)\ w)
\end{aligned}
$$

$$
\begin{array}{ccc}
ExpAr\ a & \xleftarrow{inExpAr} & 1 + (A + (B\ X\ (ExpAr\ a\ X\ ExpAr\ a) + C\ X\ ExpAr\ a)) \\
{\scriptstyle cataExpAr\ sd\_gen}\downarrow & & \downarrow{\scriptstyle recExpAr\ (cataExpAr\ sd\_gen)} \\
ExpAr\ a\ X\ ExpAr\ a & \xleftarrow[sd\_gen]{} & expr
\end{array}
$$

expr = 1 + (A + (B X ((ExpAr a,ExpAr a) X (ExpAr a, ExpAr a)) + C X (ExpAr a,ExpAr a)))

Por fim, na alínea 5, de forma semelhante à alínea 4, também foi mantido um catamorfismo que retorna um par, em que o primeiro elemento é o valor calculado substituindo por a e o segundo elemento é o valor derivado substituindo por a.

$$
\begin{aligned}
&ad\_gen\ a = [f, g] \\
&\quad \textbf{where } f = \underline{(a, 1)} \\
&\qquad g = [h, j] \\
&\qquad h\ (b) = (b, 0) \\
&\qquad j = [k, l] \\
&\qquad k\ (Sum, ((z, w), (x, y))) = (z + x, w + y) \\
&\qquad k\ (Product, ((z, w), (x, y))) = (z * x, z * y + w * x) \\
&\qquad l\ (Negate, (z, w)) = (-z, -w) \\
&\qquad l\ (E, (z, w)) = (expd\ z, w * expd\ z)
\end{aligned}
$$

$$
\begin{array}{ccc}
ExpAr\ a & \xleftarrow{inExpAr} & 1 + (A + (B\ X\ (ExpAr\ a\ X\ ExpAr\ a) + C\ X\ ExpAr\ a)) \\
{\scriptstyle cataExpAr\ ad\_gen}\downarrow & & \downarrow{\scriptstyle recExpAr\ (cataExpAr\ ad\_gen)} \\
\mathbb{N}_0\ X\ \mathbb{N}_0 & \xleftarrow[ad\_gen]{} & 1 + (A + (B\ X\ ((\mathbb{N}_0, \mathbb{N}_0)\ X\ (\mathbb{N}_0, \mathbb{N}_0)) + C\ X\ (\mathbb{N}_0, \mathbb{N}_0)))
\end{array}
$$

Problema 2

Definir

$$
\begin{aligned}
&loop\ (tot, up, down, up\_inc, down\_inc) \\
&\quad = (up \div down, up * (up\_inc + 1) * up\_inc, down * (down\_inc + 1) * down\_inc, up\_inc + 2, down\_inc + 1) \\
&inic = (1, 1, 1, 3, 2) \\
&prj\ (a, b, c, d, e) = a
\end{aligned}
$$

por forma a que

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

seja a função pretendida.

Para resolver o problema 2, o problema foi abordado da seguinte forma: tendo (2n)! no numero e n! e (n+1)! no denominador, sempre que realizar um produto em baixo, 2 produtos terão que ser feitos em cima.
Por exemplo:

((2n x (2n-1)) X (((2n-2) x (2n-3)) X ... X 4 x 3

dividirá por

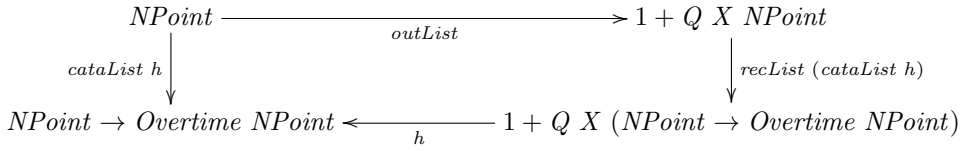
((n+1) x n)) X (n x (n-1))) X ... X 3 x 2

Assim foi criada um função up , responsável pelos produtos no numerador e a função down responsável pelos produtos no denominador. *DownInc* e *upInc* permitem ter disponíveis os valores a multiplicar no momento.

Problema 3

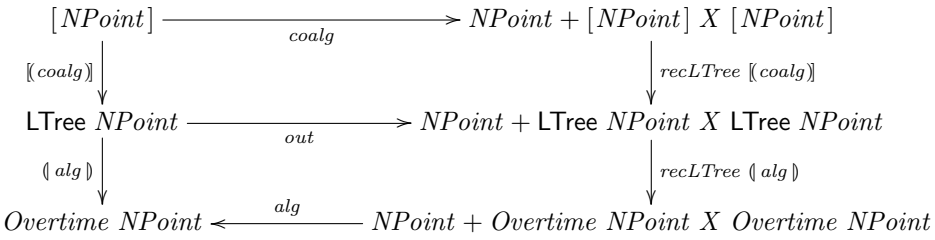
```
calcLine :: NPoint → (NPoint → OverTime NPoint)
calcLine = cataList h where
  h = [i, g]
  i l = nil
  g :: (ℚ, NPoint → OverTime NPoint) → (NPoint → OverTime NPoint)
  g (d, f) l = case l of
    [] → nil
    (x : xs) → λz → concat $ (sequenceA [singl · linear1d d x, f xs]) z
deCasteljau :: [NPoint] → OverTime NPoint
deCasteljau = hyloAlgForm alg coalg where
  coalg [] = i1 ([])
  coalg [p] = i1 (p)
  coalg l = i2 (init l, tail l)
  alg = [f, g]
  f [] = nil
  f p = p
  g ((op1, op2)) = λpt → (calcLine (op1 pt) (op2 pt)) pt
hyloAlgForm = hyloLTree
```

Sendo o *calcLine* o catamorfismo de um dado gene h, o essencial era definir este gene. Para isso, foi necessário basear a solução no seguinte diagrama.



Desta forma, o gene h dividia-se no caso nulo e no caso Q X (NPoint -¿ Overtime NPoint).
ambos definidos na função disponibilizada no enunciado e sendo só necessário adaptar.

Para a definição do *deCasteljau*, identificou-se que a estrutura intermediária apropriada para o hilomorfismo seria uma Leaf tree. Usando esta estrutura, era possível partir a lista num ponto e em duas listas de pontos, que depois do anamorfismo tornam a execução do catamorfismo mais eficiente , devido ao tipo NPoint + LTree NPoint x LTree NPoint . O catamorfismo segue o mesmo algoritmo aplicado na função disponibilizada.



Problema 4

Solução para listas não vazias:

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

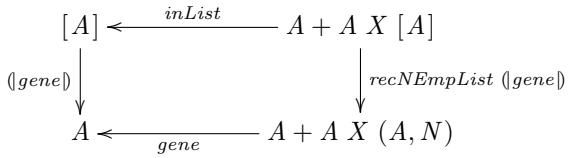
$$\begin{aligned}
&inNEmpList = [singl, cons]\\
&outNEmpList\ [a] = i_1\ (a)\\
&outNEmpList\ (a : x) = i_2\ (a, x)\\
&recNEmpList\ f = id + id \times f\\
&cataNEmpList\ g = g \cdot recNEmpList\ (cataNEmpList\ g) \cdot outNEmpList\\
&avg_aux = cataNEmpList\ (gene)\\
&\quad \textbf{where}\ gene = [b, q]\\
&\qquad b = \langle id, \underline{1} \rangle\\
&\qquad q = \langle avg, length \rangle\\
&\qquad avg\ (a, (x, y)) = (a + x * y)\ /\ (y + 1)\\
&\qquad length\ (_, (_, y)) = y + 1
\end{aligned}$$

Para a alínea 1 foi necessário definir o catamorfismo para listas não vazias.

Para o *avgAaux* relembro que o tipo da List é A + A x List a. Então no gene, primeiro separamos nos 2 casos com um either b q.

- o b representa o tipo A isolado, o nosso caso de paragem. Quando chegarmos a este caso vamos querer adicionar o valor do elemento da lista e sinalizar +1 para o comprimento da lista, guardando isso num par através de um split.
- o q representa o tipo A x List a. Então dividimos o q em 2 tarefas: calcular a média num lado, atualizar o comprimento do outro.

O avg e o length recebem argumentos do tipo (a, (b,c)) em que o 'a' é o valor do nodo em que estamos atualmente e o par (b,c) é o que já se calculou recursivamente. Então pela fórmula, a média dá-se daquela fórmula e incrementa-se um ao comprimento da lista.



Solução para árvores de tipo LTree:

$$\begin{aligned}
&avgLTree = \pi_1 \cdot (\!|\ gene\ |\!)\\
&\quad \textbf{where}\ gene = [f, g]\\
&\qquad f = \langle id, \underline{1} \rangle\\
&\qquad g = \langle avg, length \rangle\\
&\qquad avg\ ((x, y), (z, w)) = (x * y + z * w)\ /\ (y + w)\\
&\qquad length\ ((_, y), (_, w)) = y + w
\end{aligned}$$

De forma semelhante à alínea 1, agora o tipo da LTree é

$$\mathbf{data}\ \mathsf{LTree}\ a = Leaf\ a \mid Fork\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a)$$

. Temos agora algo do tipo A + LTree x LTree . Então o nosso catamorfismo também vai gerar um par com a média de um lado e o número de nodos com valor do outro mas, agora nós so temos valores nas folhas.
Desta forma, dividimos em 2 partes com um either f g.
- o f terá o tipo A, e neste caso vamos querer adicionar o valor a e sinalizar um aumento de comprimento para 1.
- o g terá o tipo LTree x LTree e nesses casos vamos só querer ter atenção ao que foi encontrado recursivamente nas folhas e juntar.
Então g dá split em avg e length também para gerar o par e ambos têm tipo ((a,b), (c,d)) em que (a,b) é o par (avg,length) da LTree a da esquerda e o par (c,d) é o par da direita, obtidos recursivamente.
Vamos querer juntá-los ponderadamente com o comprimento calculados e juntar os comprimentos.

$$\begin{array}{ccc} \mathsf{LTree}\ a & \xleftarrow{inLTree} & A + Ltree\ a\ X\ \mathsf{LTree}\ a \\ {\scriptstyle (\!|gene|\!)}\downarrow & & \downarrow{\scriptstyle recLTree\ (\!|gene|\!)} \\ A & \xleftarrow[gene]{} & A + (A, N)\ X\ (A, N) \end{array}$$

Problema 5

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

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.