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

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	28
a93220	Alexandre Flores
a93294	Miguel Gomes
a90468	Rui Armada
a67674	Tiago Sousa

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 D com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

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

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

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

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

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

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

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

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

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

3.1 Stack

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

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

Pode aceder ao GHCi utilizando o comando:

```
stack ghci
```

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

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

Problema 1

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

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

- *Symbolic differentiation*
- *Automatic differentiation*

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

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

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

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

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 12 esta função está expressa como um catamorfismo. Defina o respectivo gene e, de seguida, teste as propriedades:

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

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

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 12 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 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 [?] e tópico Recursividade mútua nos vídeos das aulas teóricas.

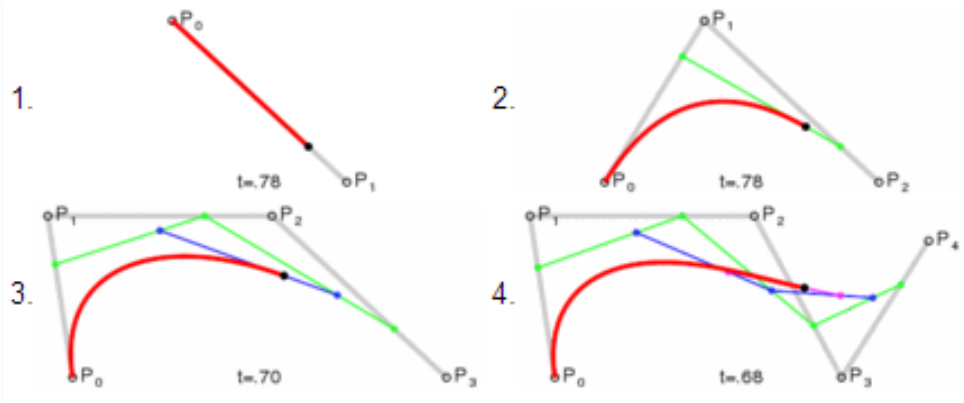

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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 C 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:*

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

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 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]

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

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

$$
\begin{array}{rr}
@C = 2cm \mathbb{N}_0[d]\_(\!|g|\!) & 1 + \mathbb{N}_0[d]^{id+(\!|g|\!)}[l]\_ \mathsf{in} \\
B & 1 + B[l]^{-}g
\end{array}
$$

(4)

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:

$$exp\ x \quad = \quad \sum_{i=0}^{\infty} \frac{x^i}{i!} \tag{5}$$

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 [?].
[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 (1):

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

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:

A função de out deste tipo deve receber uma ExpAr e fornecer os dados relevantes identificando-os apropriadamente de acordo com a inExpAr já utilizada.

Assim sendo, para expressões de variáveis (X), como não temos nenhum parâmetro, devolvemos apenas '()' identificado com Left. Todos os outros tipos de expressões serão então identificadas Right e mais um (ou dois para as expressões) identificador para os diferenciar entre eles.

$$
\begin{aligned}
&outExpAr\ X = i_1\ ()\\
&outExpAr\ (N\ a) = i_2\ (i_1\ a)\\
&outExpAr\ (Bin\ op\ ex1\ ex2) = i_2\ (i_2\ (i_1\ (op, (ex1, ex2))))\\
&outExpAr\ (Un\ op\ ex) = i_2\ (i_2\ (i_2\ (op, ex)))
\end{aligned}
$$

— A função recExpAr está responsável pela recursividade de um catamorfismo. Como tal, faz sentido que ela aplique a função fornecida como argumento apenas às parte recursivas das expressões, ou seja, à expressões que estejam dentro do tipo de dados criado pelo out. Os casos em que temos expressões dentro do nosso tipo de dados são os casos das operações, nos quais a outExpAr devolve o (devidamente identificado) par de operação expressão/expressões. Sendo assim, esta função tem de aplicar ao tipo de dados a função fornecida às expressões das possíveis operações, deixando outros identificadores iguais (id). Usando a função baseExpAr, basta indicarmos onde aplicar id e onde aplicar a função fornecida.

$$
recExpAr\ f = baseExpAr\ id\ id\ id\ f\ f\ id\ f
$$

— Num catamorfismo de ExpAr, o gene vai sempre receber o tipo 1 + (B + (C + D)), em que o 1 corresponde a uma expressão de variável, B a uma constante (N 0, por exemplo), C a uma operação binária e D a uma operação unária. Sendo assim, o nosso gene de eval tem de devolver o float fornecido para variáveis, o próprio valor para constantes, e finalmente operar segundo as operações que encontrar (que definimos nas funções auxiliares) $do_b inedo_u n$.

$$
\begin{aligned}
&g\_eval\_exp\ a = [\underline{a}, [id, do\_ops]]\ \textbf{where}\\
&\quad do\_ops = [do\_bin, do\_un]\\
&\quad do\_bin :: Num\ a \Rightarrow (BinOp, (a, a)) \to a\\
&\quad do\_bin\ (Sum, pair) = \widehat{(+)}\ pair\\
&\quad do\_bin\ (Product, pair) = \widehat{(*)}\ pair\\
&\quad do\_un :: Floating\ a \Rightarrow (UnOp, a) \to a\\
&\quad do\_un\ (Negate, x) = negate\ x\\
&\quad do\_un\ (E, x) = exp\ x
\end{aligned}
$$

— Para este hilomorfismo, o nosso anamorfismo vai tratar de limpar possíveis casos de absorção nas operações das expressões fornecidas, sendo assim responsável pela parte do algoritmo de otimização. O catamorfismo utilizado é o que definimos na alínea anterior, sendo responsável por calcular o valor de uma expressão.

$$
\begin{aligned}
&clean :: (Floating\ a, Eq\ a) \Rightarrow ExpAr\ a \to () + (a + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a)))\\
&clean = outExpAr \cdot clean\_aux\ \textbf{where}\\
&\quad clean\_aux :: (Floating\ a, Eq\ a) \Rightarrow ExpAr\ a \to ExpAr\ a\\
&\quad clean\_aux\ (Bin\ Product\ (N\ 0)\ \_) = N\ 0\\
&\quad clean\_aux\ (Bin\ Product\ \_\ (N\ 0)) = N\ 0\\
&\quad clean\_aux\ (Un\ E\ (N\ 0)) = N\ 1\\
&\quad clean\_aux\ e = e\\
&\quad --\\
&gopt\ a = g\_eval\_exp\ a
\end{aligned}
$$

A função sd já implementada executa um catamorfismo que devolve um par e usa a função p2 para usar apenas o segundo elemento do par. O par contém no elemento da esquerda a expressão original e no elemento da direita a expressão derivada. Como tal, o nosso gene terá de devolver esse par. Sendo assim, o nosso gene será um split. À esquerda do split será trabalhado o tipo de dados para remover a informação relativa à derivada (Ou seja, serão trabalhadas as expressões para que tenham apenas a informação relativa à expressão original). Após esta tarefa, o tipo de dados passa pelo inExpAr e é transformado de volta numa expressão. O lado direito do split fica então responsável por aplicar as leis da derivação aos casos diferentes de expressões, utilizando onde necessária a informação relativa às expressões originais ou às derivadas de expressões em operações calculadas recursivamente.

$sd\_gen :: (Floating\ a) \Rightarrow$
$\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \rightarrow (ExpAr\ a$
$sd\_gen = \langle inExpAr \cdot og, der \rangle$ **where**
$\quad og = id + (id + ((id \times \langle \pi_1 \cdot \pi_1, \pi_1 \cdot \pi_2 \rangle) + (id \times \pi_1)))$
$\quad\quad$ -- der = either (const (N 1)) (either (const (N 0)) der$_a ux$)
$\quad\quad$ -- der$_a ux = inExpAr.(either(i2.i2.i1.(id >< (split(p2.p1)(p2.p2))))(i2.i2.i2.(id >< p2)))$
$der :: (Floating\ a) \Rightarrow () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpA$
$der = [inExpAr \cdot i_2 \cdot i_1 \cdot \underline{1}, [\underline{(N\ 0)}, [bin\_aux, un\_aux]]]$
$bin\_aux :: (Floating\ a) \Rightarrow (BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) \rightarrow ExpAr\ a$
$bin\_aux\ (Sum, ((o1, d1), (o2, d2))) = (Bin\ Sum\ d1\ d2)$
$bin\_aux\ (Product, ((o1, d1), (o2, d2))) = (Bin\ Sum\ (Bin\ Product\ o1\ d2)\ (Bin\ Product\ d1\ o2))$
$un\_aux :: (Floating\ a) \Rightarrow (UnOp, (ExpAr\ a, ExpAr\ a)) \rightarrow ExpAr\ a$
$un\_aux\ (E, (o, d)) = (Bin\ Product\ (Un\ E\ o)\ d)$
$un\_aux\ (Negate, (o, d)) = (Un\ Negate\ d)$

A função ad, tal como a sd, usa um catamorfismo que devolve um par e apenas usa o segundo elemento do par. Este par contém na esquerda o valor da expressão original e na direita o valor da expressão derivada. Como tal, o nosso gene vai ser um split em que à esquerda trabalhamos com os valores calculados recursivamente para a expressão original, e à direita aplicamos as leis da derivação relevantes para calcular a derivada usando valores calculados recursivamente nas expressões de operações. As funções auxiliares x$_a dtrabalhamtodassobreumtipodeexpressão, aplicandoasleisdaderivaçãorelevantesparacadatipo.$

$ad\_gen :: Floating\ a \Rightarrow a \rightarrow () + (a + ((BinOp, ((a, a), (a, a))) + (UnOp, (a, a)))) \rightarrow (a, a)$
$ad\_gen\ v = [var\_ad\ v, [const\_ad, [bin\_ad, un\_ad]]]$
$var\_ad :: Floating\ a \Rightarrow a \rightarrow () \rightarrow (a, a)$
$var\_ad\ v = \langle \underline{v}, \underline{1} \rangle$
$const\_ad :: Floating\ a \Rightarrow a \rightarrow (a, a)$
$const\_ad = \langle id, \underline{0} \rangle$
$bin\_ad :: Floating\ a \Rightarrow (BinOp, ((a, a), (a, a))) \rightarrow (a, a)$
$bin\_ad\ (Sum, ((o1, d1), (o2, d2))) = ((o1 + o2), (d1 + d2))$
$bin\_ad\ (Product, ((o1, d1), (o2, d2))) = ((o1 * o2), (o1 * d2 + d1 * o2))$
$un\_ad :: Floating\ a \Rightarrow (UnOp, (a, a)) \rightarrow (a, a)$
$un\_ad\ (Negate, (o, d)) = ((-o), (-d))$
$un\_ad\ (E, (o, d)) = ((exp\ o), ((exp\ o) * d))$

Problema 2

Incrementa os denominaodes. O caso de paragem do d é 1 e o caso de paragem desta função dá 2 porque estamos a trabalhar no padrão n+1, ou seja, quando fazemos 1 (caso paragem do d) + 1 dá 2 que corresponde ao caso de paragem do di. Aqui estamos a somar 1 porque estamos no padrão n+1 porque é como se tivessemos a somar começando no 1 (sem ser no caso de paragem), ou seja, 0+1 fica 1 por isso corresponde a 1 + numei.

$denominator\_incrementer\ 0 = 2$
$denominator\_incrementer\ (n + 1) = 1 + denominator\_incrementer\ n$

Denominador

$denominator\ 0 = 1$
$denominator\ (n + 1) = denominator\_incrementer\ n * denominator\ n$

Incrementa os numeradores sucessivos. O caso de paragem do nume é 1 e o caso de paragem desta função dá 3 porque estamos a trabalhar no padrão n+1, ou seja, quando fazemos 1 (caso paragem do nume) + (1+1) dá 3 que corresponde ao caso de paragem do numei. Aqui estamos a somar 2 porque estamos no padrão n+1 porque é como se tivessemos a somar começando no 1 (sem ser no caso de paragem), ou seja, 1+1 fica 2 por isso corresponde a 2 + numei.

$$numerator\_incrementer\ 0 = 3$$
$$numerator\_incrementer\ (n+1) = 2 + numerator\_incrementer\ n$$

Numerador

$$numerator\ 0 = 1$$
$$numerator\ (n+1) = numerator\_incrementer\ n * numerator\ n$$

Fração que corresponde ao quociente entre o numerador e o denominador

$$quotient\ 0 = 1$$
$$quotient\ n = numerator\ n\ /\ denominator\ n$$

Na primeira componente temos o quociente entre o numero e o denominador. Na segunda compoente temos o produto do numerador pelo seu numerador sucessivo + 1 pelo numerador sucessivo para obter o numerador pretendido. Na terceira compoente temos o produto do denominador pelo seu denominadorsucessivo + 1 pelo denominador sucessivo para obter o numerador pretendido. A quarta componente corresponde à função numei para incrementar os numeradores sucessivos. A quinta componente corresponde à função a para incrementar os denominadores sucessivos.

$$loop\ (quotient, numerator, denominator, numerator\_incrementer, denominator\_incrementer) = (numerador \div de$$

Valores dos casos de paragem de cada uma das funções

$$inic = (1, 1, 1, 3, 2)$$

Função principal é a f

$$prj\ (f, g, h, i, j) = f$$

por forma a que

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

seja a função pretendida. **NB**: usar divisão inteira.

Problema 3

A resoluão da primeira alínea passar por adaptarmos o algoritmo fornecido a um catamorfismo. Usando a função auxiliar g usada na função fornecida, basta criarmos um catamorfismo de listas que aplica essa função auxiliar a listas não vazias e devolve a lista vazia para listas também vazias (Ou seja, pontos de dimensão 0).

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

Para a segunda alínea, decidimos utilizar um hilomorfismo de LTree. O anamorfismo será responsável por, para listas de NPoints não vazias (sendo o caso da lista) vazia tratada por pattern matchin na função deCasteljau), gerar uma LTree em que o ramo da esquerda corresponde ao init da lista e o ramo da direita corresponde à tail da lista.

Este é o passo de "divide"do nosso hilomorfismo, em que dividimos a lista de pontos até só termos um ponto em cada nodo.

O nosso catamorfismo é assim a fase de conquer, em que resolvemos os problemas que foram reduzidos no divide. Usamos a função auxiliar utilizada na função deCasteljau fornecida para transformar os pontos nos pares de leafs das ltree, transformando gradualmente a LTree num OverTime NPoint através do cálculo de baixo para cima das curvas.

$$
\begin{aligned}
&outNList :: [a] \to a + (a, [a])\\
&outNList\ [a] = i_1\ a\\
&outNList\ (a : x) = i_2\ (a, x)\\
&deCasteljau :: [NPoint] \to OverTime\ NPoint\\
&deCasteljau\ [\,] = nil\\
&deCasteljau\ l = (hyloAlgForm\ alg\ coalg)\ l\ \mathbf{where}\\
&\quad coalg = ((id) + \langle init \cdot cons, \pi_2 \rangle) \cdot outNList\\
&\quad alg = [\underline{\cdot}, g\_aux]\\
&\quad g\_aux :: (OverTime\ NPoint, OverTime\ NPoint) \to OverTime\ NPoint\\
&\quad g\_aux\ (ot1, ot2) = \lambda pt \to (calcLine\ (ot1\ pt)\ (ot2\ pt))\ pt\\
&hyloAlgForm\ f\ g = (\!|\, f \,|\!) \cdot [\!(g)\!]
\end{aligned}
$$

Problema 4

Solução para listas não vazias:

Para a resolução da primeira alínea, resolvemos o problema utilizando a lei da recursividade mútua (*Fokkinga*). Assim sendo, desenvolvemos da seguinte forma:

$avg\_aux = <avg, len>$

Logo, introduzindo o catamorfismo...

$$
\begin{aligned}
& <avg, len> = (\!|\langle h, k \rangle|\!)\\
\equiv & \quad \{\ \text{Lei Fokkinga, F f = id + id x f para listas}\ \}\\
& \begin{cases} avg \cdot \mathbf{in} = h \cdot (id + id\ x < avg, len>) \\ len \cdot \mathbf{in} = k \cdot (id + id\ x < avg, len>) \end{cases}
\end{aligned}
$$

Desenvolvendo o sistema, separando o h e o k em h1 e h2, k1 e k2, e introduzindo variáveis, ficamos com as seguintes equações:

$$
\begin{cases}
\begin{cases} avg\ [\,] = 0 \\ avg\ (a : as) = (a + (avg\ as * len\ as))\ /\ (len\ as + 1) \end{cases}\\
\begin{cases} len\ [\,] = 0 \\ len\ (a : as) = 1 + len\ as \end{cases}
\end{cases}
$$

Finalmente removemos as variáveis novamente e definimos a nossa implementação efetiva em haskell.

$$
\begin{cases}
\begin{cases} avg \cdot nil = \underline{0} \\ avg \cdot cons = mydiv \cdot <myadd\ .\ <\pi_1\ (mymul \cdot \pi_2) > k2> \end{cases}\\
\begin{cases} len \cdot nil = \underline{0} \\ len \cdot cons = \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \end{cases}
\end{cases}
$$

Em que definimos, pela seguinte ordem, h1, h2, k1, e k2. Usando a lei da troca, chegamos ao formato pedido no enunciado.

$$
\begin{aligned}
& <avg, len> = (\!|<[h1, h2], [h1, h2]>|\!)\\
\equiv & \quad \{\ \text{Lei da troca}\ \}\\
& <avg, len> = (\!|[<h1, k1>, <h2, k2>]|\!)\\
\equiv & \quad \{\ \text{Definições de h e k}\ \}\\
& <avg, len> = (\!|[<\underline{0}, \underline{0}>, <mydiv\ .\ <myadd\ .\ <\pi_1, (mymul \cdot \pi_2)>, k2>, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2>]|\!)
\end{aligned}
$$

E finalmente chegamos à solução da alínea. São definidas e utilizadas as funções mydiv, etc devido a problemas com o interpretador no uso das funções já definidas nos módulos fornecidos.

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

$$mydiv = \widehat{(/)}$$
$$myadd = \widehat{(+)}$$
$$mymul = \widehat{(*)}$$
$$avg\_aux = cataList\ [\langle \underline{0}, \underline{0} \rangle, \langle h2, k2 \rangle]\ \textbf{where}$$
$$k2 = \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2$$
$$h2 = mydiv \cdot \langle myadd \cdot \langle \pi_1, mymul \cdot \pi_2 \rangle, k2 \rangle$$

Solução para árvores de tipo LTree:

Para este caso olhamos para o diagrama do catamorfismo em questão. Esta alínea passa por adaptar o nosso gene ao catamorfismo da LTree.

@C=2cm LTree $(\mathbb{N}_0)$ [d]_$(\!|gene|\!)\mathbb{N}_0 + \mathsf{LTree}\ (\mathbb{N}_0) \times \mathsf{LTree}\ (\mathbb{N}_0)[d]^{id+(\!|gene|\!)\times(\!|gene|\!)}[l]_$ **in**
$(A, A)\mathbb{N}_0 + (A, A) \times (A, A)[l]^{-}gene = [<h1, k1>, <h2, k2>]$

Definimos então h1 e k1 como a média e comprimento de uma Leaf e h2 e k2 como média e comprimento de um Fork, que são definições facilmente implementadas em haskell.

$$avgLTree = \pi_1 \cdot (\!|\, gene \,|\!)\ \textbf{where}$$
$$k2 = myadd \cdot \langle \pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2 \rangle$$
$$h2 = mydiv \cdot \langle myadd \cdot \langle mymul \cdot \pi_1, mymul \cdot \pi_2 \rangle, k2 \rangle$$
$$gene = [\langle id, \underline{1} \rangle, \langle h2, k2 \rangle]$$

Problema 5

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

```
module cp2021t

open Cp


// (1) Datatype definition ------------------------------------------------------
//data BTree a = Empty | Node(a, (BTree a, BTree a)) deriving Show
type BTree<'a> = Empty | Node of 'a * (BTree<'a> * BTree<'a>)

let inBTree a = either (konst Empty) Node x

let outBTree a =
    match a with
        | Empty -> Left ()
        | (Node(a , (t1 , t2))) -> Right (a , (t1 , t2))

// (2) Ana + cata + hylo --------------------------------------------------------
let recBTree g = baseBTree id g

let cataBTree g a = (g << (recBTree (cataBTree g)) << outBTree) a

let anaBTree g a = (inBTree << (recBTree (anaBTree g) ) << g) a

let hyloBTree h g = cataBTree h << anaBTree g

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

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

```
(*
    instance Functor BTree
            where fmap f = cataBTree ( inBTree . baseBTree f id )
*)

// (4) Examples ------------------------------------------------------------
// (4.1) Inversion (mirror) ------------------------------------------------
let invBTree a = cataBTree (inBTree << (id -|- (id >< swap))) a

// (4.2) Counting ----------------------------------------------------------

let countBTree a = cataBTree (either (konst 0) (succ << (uncurry (+)) << p2)) a

// (4.3) Serialization -----------------------------------------------------
let inordt = cataBTree inord

let inord a = either nil join

let join (x,(l,r)) = l @ [x] @ r

let preordt = cataBTree preord

let preord = (either nil preord_g)

let preord_g (x,(l,r)) = x :: l @ r

let postordt = cataBTree (either nil postordt_g)

let postordt_g (x,(l,r)) = l @ r @ [x]

// (4.4) Quicksort ---------------------------------------------------------
let qSort a = hyloBTree inord qsep

//let qsep [] = Left ()
//let qsep list = Right (list.Head part (fun a -> a < list.Head) list.Tail)

let qsep list =
    match list with
        | list.isEmpty -> Left ()
        | _            -> Right (list.Head part ( fun a -> a < list.Head)  list.Tail)

let part p [] = ([],[])
let rec part p list =
    match list.Length with
        | p list.Head -> let (s,l) = part p list.Tail in (list.Head @ s, l)
        | otherwise   -> let (s,l) = part p list.Tail in (s, list.Head @ l)

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

let traces = cataBTree (Either (konst [[]]) tunion)

let tunion (a, (l,r)) = union (map (fun x -> a @ x) l) (map (fun x -> a @ x) r)

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

let hanoi = hyloBTree present strategy

// where
```

```
let present = inord

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

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

let balBTree = p1 . baldepth

let depthBTree = p2 . baldepth

let baldepth = cataBTree depbal

let depbal = Either (konst (True, 1)) ( (fun (a,((b1,b2),(d1,d2))) -> (b1 && b2 && ab
!<  (fun ((b1,d1),(b2,d2)) -> ((b1,b2),(d1,d2))))

// (6) Going polytipic -------------------------------------------------------

// (7) Zipper ----------------------------------------------------------------
type Deriv<'a> = Dr Bool 'a * (Btree<'a>)

type Zipper a = [ Deriv a ]

let rec plug list t =
    match list with
        | list.isEmpty  -> t
        | _             -> match list.Head with
                               | Dr (False a l) -> Node (a, (plug list.Tail t, l))
                               | Dr (True a r)  -> Node (a, (r, plug list.Tail t))
```