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

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	74
a93269	Inês Oliveira Anes Vicente
a93308	Jorge Miguel Silva Melo
a93280	Miguel Ângelo Machado Martins

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.

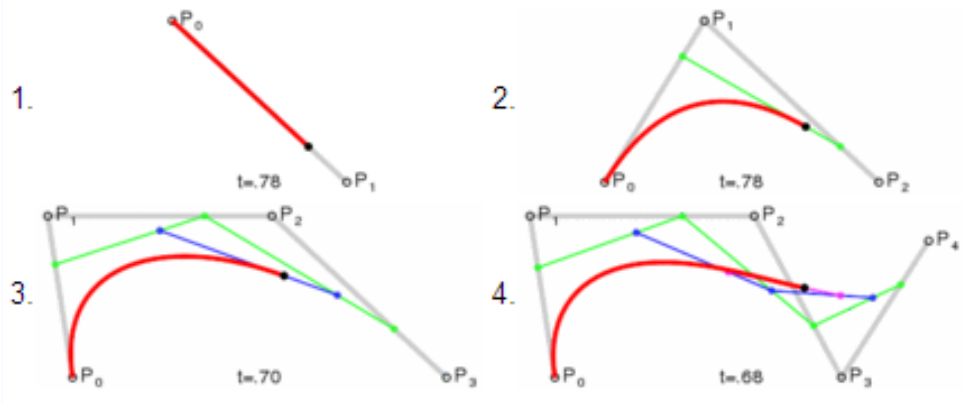

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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:

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

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.

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

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

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

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

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

O anexo C tem definida a função

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

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

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

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.*

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

2. Implemente a função $\mathit{deCasteljau}$ 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 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:

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

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 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
```

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

D.0.1 Alínea 1

Para resolver esta alínea, podemos começar por inferir o $outExpAr$ através da propriedade do isomorfismo:

$$
\begin{aligned}
& outExpAr \cdot inExpAr = id\\
\equiv & \quad \{\ inExpAr := [\underline{X}, num\_ops]\ \}\\
& outExpAr \cdot [\underline{X}, num\_ops] = id\\
\equiv & \quad \{\ \text{Fusão-+(20) \& Functor-id(26)}\ \}\\
& [outExpAr \cdot \underline{X}, outExpAr \cdot num\_ops] = [i\_1, i\_2]\\
\equiv & \quad \{\ num\_ops := [N, ops]\ \&\ ops := [bin, \widehat{Un}]\ \text{\& Eq-+(27)}\ \}\\
& \begin{cases} outExpAr \cdot \underline{X} = i\_1\\ outExpAr \cdot [N, [bin, \widehat{Un}]] = i\_2 \end{cases}\\
\equiv & \quad \{\ \text{Fusão-+(20)}\ \}\\
& \begin{cases} outExpAr \cdot \underline{X} = i\_1\\ [outExpAr \cdot N, outExpAr \cdot [bin, \widehat{Un}]] = i\_2 \end{cases}\\
\equiv & \quad \{\ \text{Universal-+(17)}\ \}\\
& \begin{cases} outExpAr \cdot \underline{X} = i\_1\\ \begin{cases} outExpAr \cdot N = i\_2 \cdot i\_1\\ outExpAr \cdot [bin, \widehat{Un}] = i\_2 \cdot i\_2 \end{cases} \end{cases}\\
\equiv & \quad \{\ \text{Fusão-+(20) \& Universal-+(17)}\ \}\\
& \begin{cases} outExpAr \cdot \underline{X} = i\_1\\ \begin{cases} outExpAr \cdot N = i\_2 \cdot i\_1\\ \begin{cases} outExpAr \cdot bin = i\_2 \cdot i\_2 \cdot i\_1\\ outExpAr \cdot \widehat{Un} = i\_2 \cdot i\_2 \cdot i\_2 \end{cases} \end{cases} \end{cases}\\
\equiv & \quad \{\ \text{Def-comp(72) \& Introdução de variáveis}\ \}\\
& \begin{cases} outExpAr\ \underline{X}\ () = i\_1\ ()\\ \begin{cases} outExpAr\ (N\ (a)) = i\_2\ (i\_1\ (a))\\ \begin{cases} outExpAr\ (bin\ (op, (a, b))) = i\_2\ (i\_2\ (i\_1\ (op, (a, b))))\\ outExpAr\ \widehat{Un}\ (op, a) = i\_2\ (i\_2\ (i\_2\ (op, a))) \end{cases} \end{cases} \end{cases}
\end{aligned}
$$

□

$$\mathbb{N}_0 \xleftarrow{\text{in}} 1 + \mathbb{N}_0$$

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

$outExpAr\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b))))$
$outExpAr\ (Un\ op\ a) = i_2\ (i_2\ (op, a)))$

Tanto no enunciado deste projeto como nas aulas os docentes iam-nos alertando para a importância de ler e analisar código. Na resolução deste exercício acabamos por perceber essa importância, pois a análise do código do anexo C e/ou dos ficheiros da pasta *src* mostraram-se essenciais na resolução dos problemas do projeto. No caso desta segunda alínea do problema 1, depois de percebermos o funcionamento da função $baseExpAr$ fornecida, o ficheiro *List.hs* ajudou bastante na resolução deste exercício.

Ao olharmos para a função $baseExpAr$ percebemos que a mesma necessita de 7 argumentos. Acabamos por perceber que 7 argumentos eram estes ao interpretar o código da *recList*.

Sendo o *recList* o *F f* das listas, fizemos a analogia para esta alínea, tendo em conta o *outExpAr* definido na alínea anterior, uma vez que este gene só será executado depois dele.

Tendo em conta que $N\ a$ representa um certo número e X uma variável, estes dois casos poderiam ser considerados como casos de paragem, uma vez que não são mais simplificáveis. Estas podem portanto ser do tipo id, fazendo aqui uma analogia clara com o caso de paragem das listas. Porém, nos outros 2 casos, a expressão pode necessitar de mais simplificações. No caso da $Un\ UnOp\ (ExpAr\ a)$ só a $ExpAr$ precisa de ser simplificada, ficando com o tipo $id \times f$, tal como acontecia, aliás, no caso das listas. Já na expressão com o operador binário, é necessário simplificar mais uma expressão do que no caso anterior, ficando, portanto, com o tipo $id \times (f \times f)$.

Relembrando a definição da função $baseExpAr$, que a a $recExpAr$ invoca:

$baseExpAr\ f\ g\ h\ j\ k\ l\ z = f + (g + (h \times (j \times k) + l \times z))$

podemos facilmente atribuir valores a f,g,h,j,k,l e z:

- f é id e representa a X.
- g é id e representa a $N\ a$.
- $h \times (j \times k)$ é $id \times (f \times f)$ e representa a $Bin\ BinOp\ (ExpAr\ a)\ (ExpAr\ a)$.
  h é, portanto, id, j e k são f.
- Por último, a expressão unária $Un\ UnOp\ (ExpAr\ a)$ é representada por $l \times z$, ou seja, $id \times f$.

A expressão final pretendida é $id + (id + (id \times (f \times f) + id \times f))$, que é facilmente inserida na função $baseExpAr$ com as conclusões enumeradas em cima.
Temos, assim, reunidas as condições para afirmar que:

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

D.0.2 Alínea 2

Para a resolução desta alínea consideramos que o gene deveria simplificar expressões, aplicando as propriedades das expressões que as mesmas representam.
O gene deve ter, portanto, 2 argumentos: os mesmos 2 da $eval\_exp$. São estes: um $Floating\ a$ e uma expressão aritmética.
Assim sendo, teremos 4 casos, estando os casos das operações binárias e unárias divididos em 2 subcasos:

$g\_eval\_exp\ x\ (i_1\ ()) = x$
$g\_eval\_exp\ x\ (i_2\ (i_1\ a)) = a$
$g\_eval\_exp\ x\ (i_2\ (i_2\ (i_1\ (op, (a, b)))))$
$\quad \mid op \equiv Sum = a + b$

$$
\begin{aligned}
&\quad \mid op \equiv Product = a * b \\
&g\_eval\_exp\ x\ (i_2\ (i_2\ (op, a)))) \\
&\quad \mid op \equiv Negate = -a \\
&\quad \mid op \equiv E = expd\ a
\end{aligned}
$$

D.0.3 Alínea 3

A Simplificação das expressões é feita na função $clean$, cujo o nome sugestivo indicava que seria aqui que a expressão seria de facto simplificada. Por outro lado, a função $gopt$ poderia reaproveitar o código da alínea anterior(da função g_eval_exp), visto que esta função já tem definido o procedimento para simplificar expressões e só é corrida depois da $clean$. Deste modo estariamos a efetuar cálculos somente se a $clean$ não tivesse tirado já proveito dos elementos absorventes das operações.
E que elementos podem ser estes?
No nosso caso identificamos 2 bastante conhecidos das propriedades matemáticas:

- No caso da operação binária da multiplicação, sabe-se que que qualquer valor/expressão multiplicad por 0(no caso do exercício $N\ 0$) resulta em 0.

- No caso da operação unária do exponencial de base *e*, sabemos que se o número de Euler/Nepper estiver elevado a 0, ou seja no caso deste exercício, a $N\ 0$, terá como resultado o valor 1.

Temos, assim, reunidas as condições para afirmar que a $clean$ de uma $ExpAr$ pode ser dada por:

$$
\begin{aligned}
&clean\ (Bin\ Product\ (N\ 0)\ b) = i_2\ (i_1\ 0) \\
&clean\ (Bin\ Product\ a\ (N\ 0)) = i_2\ (i_1\ 0) \\
&clean\ (Un\ E\ (N\ 0)) = i_2\ (i_1\ 1) \\
&clean\ exp = outExpAr\ exp
\end{aligned}
$$

A $gopt$, como foi referido, tira aproveitamento da definição da g_eval_exp:

$$
gopt\ x = g\_eval\_exp\ x
$$

D.0.4 Alínea 4

Na resolução desta alínea, o mais importante foi perceber como é que, começando da expressão aritmética, poderiamos chegar ao par de $ExpAr$'s pretendido.

Ora, para tal é também essencial compreender que a regra da derivada do produto de 2 funções não só necessita da derivada de uma função como também necessita da função em si.

Analisando código, também isso uma parte essencial deste trabalho, podemos concluir que este gene verá o segundo elemento do par a ser selecionado pelo π_2 na assinatura da função sd. Visto que o objetivo da função sd é calcular a derivada, podemos definir o seu gene como uma função que devolve o valor da função como primeiro elemento do par e o valor da derivada no segundo elemento do par.

$$
\begin{aligned}
&sd\_gen :: Floating\ a \Rightarrow \\
&\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \\
&\quad \rightarrow (ExpAr\ a, ExpAr\ a) \\
&sd\_gen\ (i_1\ ()) = (X, (N\ 1)) \\
&sd\_gen\ (i_2\ (i_1\ a)) = ((N\ a), (N\ 0)) \\
&sd\_gen\ (i_2\ (i_2\ (i_1\ (Sum, (a, b))))) = (Bin\ Sum\ (\pi_1\ a)\ (\pi_1\ b), Bin\ Sum\ (\pi_2\ a)\ (\pi_2\ b)) \\
&sd\_gen\ (i_2\ (i_2\ (i_1\ (Product, (a, b))))) = (Bin\ Product\ (\pi_1\ a)\ (\pi_1\ b), Bin\ Sum\ fst\_aux\ snd\_aux) \\
&\quad \textbf{where}\ fst\_aux = Bin\ Product\ (\pi_1\ a)\ (\pi_2\ b) \\
&\qquad snd\_aux = Bin\ Product\ (\pi_2\ a)\ (\pi_1\ b) \\
&sd\_gen\ (i_2\ (i_2\ (i_2\ (E, a)))) = (Un\ E\ (\pi_1\ a), Bin\ Product\ (Un\ E\ (\pi_1\ a))\ (\pi_2\ a)) \\
&sd\_gen\ (i_2\ (i_2\ (i_2\ (Negate, a)))) = (Un\ Negate\ (\pi_1\ a), Un\ Negate\ (\pi_2\ a))
\end{aligned}
$$

D.0.5 Alínea 5

$$
\begin{aligned}
&ad\_gen\ pnt\ (i_1\ ()) = (pnt, 1)\\
&ad\_gen\ pnt\ (i_2\ (i_1\ a)) = (a, 0)\\
&ad\_gen\ pnt\ (i_2\ (i_2\ (i_1\ (Sum, (a, b))))) = ((\pi_1\ a) + (\pi_1\ b), (\pi_2\ a) + (\pi_2\ b))\\
&ad\_gen\ pnt\ (i_2\ (i_2\ (i_1\ (Product, (a, b))))) = ((\pi_1\ a) * (\pi_1\ b), ((\pi_1\ a) * (\pi_2\ b)) + ((\pi_2\ a) * (\pi_1\ b)))\\
&ad\_gen\ pnt\ (i_2\ (i_2\ (i_2\ (E, a)))) = (expd\ (\pi_1\ a), (\pi_2\ a) * expd\ ((\pi_1\ a)))\\
&ad\_gen\ pnt\ (i_2\ (i_2\ (i_2\ (Negate, a)))) = (-(\pi_1\ a), -(\pi_2\ a))
\end{aligned}
$$

Problema 2

Primeiramente, vamos tentar buscar uma relação entre o *n*-ésimo valor de Catalon e o seu (*n+1*)-ésimo valor:

$$
\begin{aligned}
&\frac{C_{n+1}}{C_n} = \frac{\frac{(2(n+1))!}{(n+2)!(n+1)!}}{\frac{(2n)!}{(n+1)!(n)!}}\\
\equiv\\
&\frac{C_{n+1}}{C_n} = \frac{(2(n+1))!(n)!}{(n+2)!(2n)!}\\
\equiv\\
&\frac{C_{n+1}}{C_n} = \frac{(2n+2)(2n+1)(2n)!(n)!}{(n+2)(n+1)(n)!(2n)!}\\
\equiv\\
&\frac{C_{n+1}}{C_n} = \frac{(2n+2)(2n+1)}{(n+2)(n+1)}\\
\equiv\\
&\frac{C_{n+1}}{C_n} = \frac{2(n+1)(2n+1)}{(n+2)(n+1)}\\
\equiv\\
&\frac{C_{n+1}}{C_n} = \frac{4n+2}{n+2}\\
\equiv\\
&C_{n+1} = \frac{4n+2}{n+2}C_n
\end{aligned}
$$

Tendo chegado a esta expressão, podemos agora dividir a equação entre 2 outras:

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

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

$$
\begin{aligned}
&den(n) = n + 2\\
&den(0) = 2\\
&den(n{+}1) = (n+3) = 1 + den(n)
\end{aligned}
$$

Pela *regra da algibeira,* teremos:

$$loop\ (a, num, den) = (a * num\ `div`\ den, 4 + num, 1 + den)$$
$$inic = (1, 2, 2)$$
$$prj\ (a, b, c) = a$$

por forma a que

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

seja a função pretendida.

Problema 3

$$calcLine :: NPoint \to (NPoint \to OverTime\ NPoint)$$
$$calcLine\ p\ q\ d = (cataList\ h)\ \$\ (zip\ p\ q)\ \mathbf{where}$$
$$\quad h\ (i_1\ ()) = [\,]$$
$$\quad h\ (i_2\ ((a1, a2), t)) = (+\!\!+)\ (singl\ \$\ (linear1d\ a1\ a2\ d))\ t$$

$$deCasteljau :: [NPoint] \to OverTime\ NPoint$$
$$deCasteljau\ list\ time = hyloAlgForm\ alg\ coalg\ list\ \mathbf{where}$$
$$\quad coalg\ [\,] = i_1\ [\,]$$
$$\quad coalg\ [a] = i_1\ a$$
$$\quad coalg\ list = i_2\ (p, q)\ \mathbf{where}$$
$$\quad\quad p = init\ list$$
$$\quad\quad q = tail\ list$$
$$\quad alg\ (i_1\ a) = a$$
$$\quad alg\ (i_2\ (p, q)) = calcLine\ p\ q\ time$$

$$hyloAlgForm = hyloLTree$$

Problema 4

Solução para listas não vazias:

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

Para usarmos um catamorfismo para listas não vazias vamos criar uma definição que não use listas vazias:

$$out\_ex4\ [a] = i_1\ (a)$$
$$out\_ex4\ (h : t) = i_2\ (h, t)$$

$$in\_ex4 = [singl, cons]$$

$$cataL\_ex4\ g = g \cdot (recList\ (cataL\_ex4\ g)) \cdot out\_ex4$$

$$
\begin{array}{ll}
& avg\_aux = (\!|[b, q]|\!) \\
\equiv & \quad \{\ avg\_aux := \langle avg, length\rangle; b = \langle b1, b2\rangle; q = \langle q1, q2\rangle\ \} \\
& aplit\ avg\ length = (\!|[\langle b1, b2\rangle, \langle q1, q2\rangle]|\!) \\
\equiv & \quad \{\ \text{Lei da troca}\ \} \\
& \langle avg, length\rangle = (\!|\langle [b1, q1], [b2, q2]\rangle|\!) \\
\equiv & \quad \{\ \text{Fokkinga}\ \} \\
& \left\{ \begin{array}{l} f \cdot \mathbf{in} = [b1, q1] \cdot \langle f, g\rangle \\ g \cdot \mathbf{in} = [b2, q2] \cdot \langle f, g\rangle \end{array} \right. \\
\equiv & \quad \{\ \mathbf{in} := [singl, cons];\ F\ f := id + id \times f\ \} \\
& \left\{ \begin{array}{l} avg \cdot [singl, cons] = [b1, q1] \cdot (id + id \times \langle avg, length\rangle \\ length \cdot [singl, cons] = [b2, q2] \cdot (id + id \times \langle avg, length\rangle \end{array} \right. \\
\equiv & \quad \{\ \text{Fusão - + ; Absorção - +; Natural -id}\ \} \\
& \left\{ \begin{array}{l} [avg \cdot singl, avg \cdot cons] = [b1, q1 \cdot \langle \pi_1, \langle avg \cdot \pi_2, length \cdot \pi_2\rangle\rangle] \\ [length \cdot singl, length \cdot cons] = [b2, q2 \cdot \langle \pi_1, \langle avg \cdot \pi_2, length \cdot \pi_2\rangle\rangle] \end{array} \right. \\
\equiv & \quad \{\ \text{Eq - +}\ \} \\
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} avg \cdot singl = b1 \\ avg \cdot cons = q1 \cdot \langle \pi_1, \langle avg \cdot \pi_2, length \cdot \pi_2\rangle\rangle \\ length \cdot singl = b2 \\ length \cdot cons = q2 \cdot \langle \pi_1, \langle avg \cdot \pi_2, length \cdot \pi_2\rangle\rangle \end{array} \right. \end{array} \right. \\
\equiv & \quad \{\ \text{Def-comp(72) ; Introdução de variáveis ; Def - x ; Def-split ; Def-singl ; Def-cons}\ \} \\
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} avg\ [x] = b1\ (x, xs) \\ avg\ (x : xs) = q1\ (\pi_1\ (x, xs), (avg\ (\pi_2\ (x, xs)), (length\ (\pi_2\ (x, xs))))) \\ length\ [x] = b2\ (x, xs) \\ length\ (x : xs) = q2 \cdot (\pi_1\ (x, xs), (avg\ (\pi_2\ (x, xs)), (length\ (\pi_2\ (x, xs))))) \end{array} \right. \end{array} \right. \\
\equiv & \quad \{\ \text{Def-proj ; avg[x] = x; length[x] = 1; length(x:xs) = succ . length(xs); Def-avg}\ \} \\
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} x = b1 \\ ((x + length * avg\ (xs))\ /\ (length + 1)) = q1\ (x, (avg\ (xs), (length\ (xs)))) \\ 1 = q1 \\ \mathsf{succ}\ \cdot length\ (xs) = q2\ (x, (avg\ (xs), (length\ (xs)))) \end{array} \right. \end{array} \right. \\
\equiv & \quad \{\ \text{Simplificação}\ \} \\
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} b1 = id \\ q1\ (x, (avg\ (xs), (length\ (xs)))) = ((x + length\ (xs) * avg\ (xs))\ /\ (length\ (xs) + 1)) \\ b2 = 1 \\ q2 = \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \end{array} \right. \end{array} \right. \\
\Box &
\end{array}
$$

Substindo os valores na expressão inicial e para código Haskell temos:

$$
\begin{array}{l}
avg\_aux :: [Double] \to (Double, Double) \\
avg\_aux = cataL\_ex4\ gene\ \mathbf{where} \\
\quad gene = [\langle id, \underline{1}\rangle, \langle aux\_split, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2\rangle] \\
\quad aux\_split\ (h, (a, l)) = (h + (l * a))\ /\ (l + 1)
\end{array}
$$

Solução para árvores de tipo LTree:

$$
\begin{aligned}
& avg\_aux = (\!|[b, q]|\!) \\
\equiv & \quad \{\ avg\_aux := \langle avg, length\rangle; b = \langle b1, b2\rangle; q = \langle q1, q2\rangle\ \} \\
& aplit\ avg\ length = (\!|[\langle b1, b2\rangle, \langle q1, q2\rangle]|\!) \\
\equiv & \quad \{\ \text{Lei da troca}\ \} \\
& \langle avg, length\rangle = (\!|\langle [b1, q1], [b2, q2]\rangle|\!) \\
\equiv & \quad \{\ \text{Fokkinga}\ \} \\
& \begin{cases} f \cdot \mathbf{in} = [b1, q1] \cdot \langle f, g\rangle \\ g \cdot \mathbf{in} = [b2, q2] \cdot \langle f, g\rangle \end{cases} \\
\equiv & \quad \{\ \mathbf{in} := [Leaf, Fork];\ F\ f := \mathrm{id} + \mathrm{f}^2\ \} \\
& \begin{cases} avg \cdot [Leaf, Fork] = [b1, q1] \cdot (id + \langle avg, length\rangle \times \langle avg, length\rangle) \\ length \cdot [Leaf, Fork] = [b2, q2] \cdot (id + \langle avg, length\rangle \times \langle avg, length\rangle) \end{cases} \\
\equiv & \quad \{\ \text{Fusão - + ; Absorção - +; Natural -id; Eq - +}\ \} \\
& \begin{cases} \begin{cases} avg \cdot Leaf = b1 \\ avg \cdot Fork = q1 \cdot (\langle avg, length\rangle \times \langle avg, length\rangle) \end{cases} \\ \begin{cases} length \cdot Leaf = b2 \\ length \cdot Fork = q2 \cdot (\langle avg, length\rangle \times \langle avg, length\rangle) \end{cases} \end{cases} \\
\equiv & \quad \{\ \text{Introdução de variáveis; Def-comp; Def-x; Def-split}\ \} \\
& \begin{cases} \begin{cases} avg\ (Leaf\ (x)) = b1\ (x) \\ avg\ (Fork\ ((x, xs), (y, ys))) = q1\ ((avg\ (x, xs), length\ (x, xs)), (avg\ (y, ys), length\ (y, ys))) \end{cases} \\ \begin{cases} length\ (Leaf\ (x)) = b2\ (x) \\ length\ (Fork\ ((x, xs), (y, ys))) = q2\ ((avg\ (x, xs), length\ (x, xs)), (avg\ (y, ys), length\ (y, ys))) \end{cases} \end{cases} \\
\equiv & \quad \{\ \text{Def-avg; Def-length; Simplificação}\ \} \\
& \begin{cases} \begin{cases} b1 = x \\ q1\ ((a1, a2), (b1, b2)) = (a1 * a2 + b1 * b2)\ /\ (a2 + b2) \end{cases} \\ \begin{cases} b2 = 1 \\ q2 = \widehat{(+)} \cdot \langle \pi_2 \cdot \pi_1, \pi_2 \cdot \pi_2\rangle \end{cases} \end{cases}
\end{aligned}
$$

Substindo os valores na expressão inicial e para código Haskell temos:

$$
\begin{aligned}
& avgLTree :: \mathsf{LTree}\ Double \to Double \\
& avgLTree = \pi_1 \cdot (\!|\ gene\ |\!) \\
& gene = [\langle id, \underline{1}\rangle, \langle aux\_split, f\rangle]\ \mathbf{where} \\
& \quad f = \widehat{(+)} \cdot \langle \pi_2 \cdot \pi_2, \pi_2 \cdot \pi_1\rangle \\
& \quad aux\_split\ ((a1, l1), (a2, l2)) = (a1 * l1 + a2 * l2)\ /\ (l1 + l2)
\end{aligned}
$$

Problema 5

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

```
module BTree

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 (a,(t1,t2)) -> i2 (a,(t1,t2))

// (2) Ana + cata + hylo ------------------------------------------------------

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

let recBTree g = baseBTree id g

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

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

let hyloBTree h g = cataBTree h << anaBTree g

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

//instance Functor BTree
//         where fmap f = cataBTree ( inBTree . baseBTree f id )
let fmap f = cataBTree ( inBTree << baseBTree f id )

// (4) Examples ---------------------------------------------------------------

// (4.0) Inversion (mirror) ---------------------------------------------------

let invBTree x = cataBTree (inBTree << (id -|- (id >< swap))) x

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

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

// (4.3) Serialization --------------------------------------------------------

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

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 // pre-order traversal

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

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

let rec part p x =
    match x with
    | [] -> ([],[])
    | (h::t) -> if (p h) then let (s,l) = part p t in ((h::s),l) else let (s,l) = par
```

```
let less h x = (if (x < h) then true else false)

let qsep x =
    match x with
    | [] -> i1()
    | (h::t) -> let (s,l) = part (less h) t in i2 (h,(s,l))

let qSort x = hyloBTree inord qsep x

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

let rec init x =
    match x with
    | [] -> []
    | (h::t) -> [h] @ (init t)

let rec last x =
    match x with
    | [a] -> a
    | (h::t) -> last t

let rec isOnList x =
    match x with
    | (b , []) -> false
    | (b , (h::t)) -> if (b = h) then true else isOnList (b , t)

let rec union x =
    match x with
    | ([] , a) -> a
    | (a , []) -> a
    | (a , b) -> if (isOnList ((last b) , a)) then (union (a , (init b))) else ((unio

let headbtl a l = (a::l)

let tunion (a,(l,r)) = union ((List.map (headbtl a) l) , (List.map (headbtl a) r))

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

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

// pointwise:
// hanoi(d,0) = []
// hanoi(d,n+1) = (hanoi (not d,n)) ++ [(n,d)] ++ (hanoi (not d, n))

let strategy x =
    match x with
    | (d,0) -> i1 ()
    | (d,n) -> i2 ((n,d),((not d,(n-1)),(not d,(n-1))))

let present x = inord x

let hanoi x = hyloBTree present strategy x

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

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

let balBTree x = (p1 << baldepth) x

let depthBTree x = (p2 << baldepth) x

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

// natural transformation from base functor to monoid
//let tnat f =
//    let theta = uncurry (<>)
//    in either (konst mempty) (theta << (f >< theta))
//
// monoid reduction
//
//let monBTree f = cataBTree (tnat f)

// alternative to (4.2) serialization --------------------------------------

//let preordt' x = monBTree singl x

// alternative to (4.1) counting -------------------------------------------

//let countBTree' x = monBTree (konst (Sum 1)) x

// (7) Zipper --------------------------------------------------------------

//type Deriv<'a> = Dr of Bool 'a BTree<'a>
//
//type Zipper<'a> = [ Deriv<'a> ]
//
//let rec plug x t =
//    match x with
//    | [] -> t
//    | ((Dr false a l)::z) = Node (a,(plug z t,l))
//    | ((Dr true a r)::z) = Node (a,(r,plug z t))
//
//------------------------- end of library ---------------------------------
```

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.