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

Departamento de Informática
Universidade do Minho

Junho de 2021

Grupo nr.	95
a93325	Henrique Costa
a93163	José Pedro Fernandes
a93246	Matilde Bravo
a93272	Pedro Alves

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 dependências externas serão instaladas automaticamente.

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

Problema 1

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

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

- *Symbolic differentiation*
- *Automatic differentiation*

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

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

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

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

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

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

O construtor E simboliza o exponencial de base e.

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

$$\mathit{Bin}\ \mathit{Sum}\ X\ (N\ 10)$$

designa $x + 10$ na notação matemática habitual.

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

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

Defina as funções $\mathit{outExpAr}$ e $\mathit{recExpAr}$, e teste as propriedades que se seguem.

Propriedade [QuickCheck] 1 *$\mathit{inExpAr}$ e $\mathit{outExpAr}$ são testemunhas de um isomorfismo, isto é, $\mathit{inExpAr} \cdot \mathit{outExpAr} = \mathit{id}$ e $\mathit{outExpAr} \cdot \mathit{idExpAr} = \mathit{id}$:*

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

- Regra do produto:

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

Defina o gene do catamorfismo que ocorre na função

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

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

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

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

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

   Defina o gene do catamorfismo que ocorre na função

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

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

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

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

Problema 2

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

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

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

[4] Lei (3.94) em [?], página 98.

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

Obter-se-á de imediato

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

usando as regras seguintes:

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

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

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

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

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

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

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

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

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

que implemente esta função.

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

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

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

Problema 3

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

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

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

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

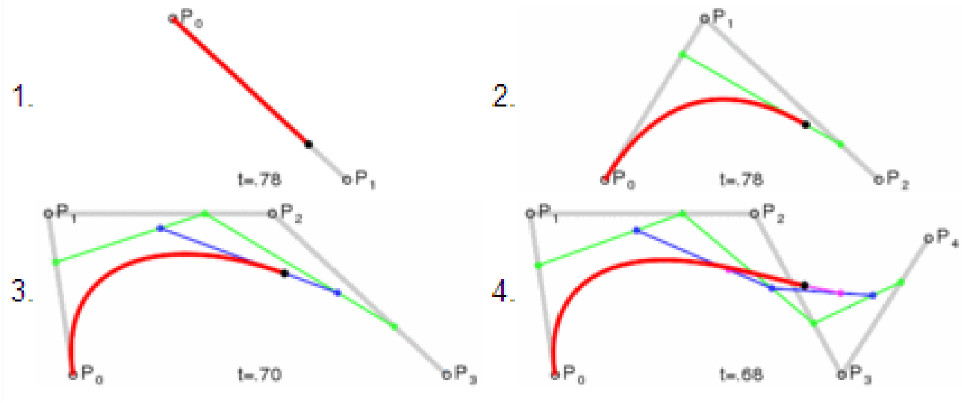

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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

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

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

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

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

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

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

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

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

$$\mathbf{type}\ OverTime\ a = Float \rightarrow a$$

O anexo C tem definida a função

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

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

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

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.*

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

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

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

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

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

Problema 4

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

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

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

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

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

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

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

Verifique as suas funções testando a propriedade seguinte:

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

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

Problema 5

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

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

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

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

Anexos

A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

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

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

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

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

B Programação dinâmica por recursividade múltipla

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

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

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

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

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

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

[8] Exemplos tirados de [?].

[9] Cf. [?], página 102.

C Código fornecido

Problema 1

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

Problema 2

Definição da série de Catalan usando factoriais (1):

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

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

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

Problema 3

Algoritmo:

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

Função auxiliar:

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

2D:

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

Modelo:

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

[10] Fonte: Wikipedia.

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

Gloss:

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

Animação:

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

Propriedades e *main*:

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

Compilação e execução dentro do interpretador:[11]

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

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

QuickCheck

Código para geração de testes:

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

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

Outras funções auxiliares

Lógicas:

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

D Soluções dos alunos

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

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

Problema 1

São dadas:

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

$$eval\_exp :: Floating\ a \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$$
$$eval\_exp\ a = cataExpAr\ (g\_eval\_exp\ a)$$
$$optmize\_eval :: (Floating\ a, Eq\ a) \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$$
$$optmize\_eval\ a = hyloExpAr\ (gopt\ a)\ clean$$
$$sd :: Floating\ a \Rightarrow ExpAr\ a \rightarrow ExpAr\ a$$
$$sd = \pi_2 \cdot cataExpAr\ sd\_gen$$
$$ad :: Floating\ a \Rightarrow a \rightarrow ExpAr\ a \rightarrow a$$
$$ad\ v = \pi_2 \cdot cataExpAr\ (ad\_gen\ v)$$

Definir:

outExpAr

A implementação desta função é deduzida a partir da própria propriedade enunciada:
outExpAr . inExpAr .==. id

Deste modo, para descobrirmos **outExpAr** basta fazermos sua composição com **inExpAr** e igualarmos ao **id**. Aplicando algumas propriedades, temos:

$$outExpAr \cdot inExpAr = id$$
$$\equiv \quad \{ \text{ def-inExpAr } \}$$
$$outExpAr \cdot [\underline{X}, num\_ops] = id$$
$$\equiv \quad \{ \text{ fusão-+ } \}$$
$$[outExpAr \cdot \underline{X}, outExpAr \cdot num\_ops] = id$$
$$\equiv \quad \{ \text{ Universal-+, Natural-id } \}$$
$$\begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot num\_ops = i_2 \end{cases}$$
$$\equiv \quad \{ \text{ def-}num\_ops \}$$
$$\begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot [N, ops] = i_2 \end{cases}$$
$$\equiv \quad \{ \text{ fusão-+ } \}$$
$$\begin{cases} outExpAr \cdot \underline{X} = i_1 \\ [outExpAr \cdot N, outExpAr \cdot ops] = i_2 \end{cases}$$
$$\equiv \quad \{ \text{ Universal-+ } \}$$
$$\begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot ops = i_2 \cdot i_2 \end{cases}$$
$$\equiv \quad \{ \text{ def-ops } \}$$
$$\begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot [bin, \widehat{Un}] = i_2 \cdot i_2 \end{cases}$$
$$\equiv \quad \{ \text{ fusao-+ } \}$$
$$\begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ [outExpAr \cdot bin, outExpAr \cdot \widehat{Un}] = i_2 \cdot i_2 \end{cases}$$
$$\equiv \quad \{ \text{ universal-+ } \}$$

$$\begin{cases} outExpAr \cdot \underline{X} = i_1 \\ outExpAr \cdot N = i_2 \cdot i_1 \\ outExpAr \cdot bin = i_2 \cdot i_2 \cdot i_1 \\ outExpAr \cdot \widehat{Un} = i_2 \cdot i_2 \cdot i_2 \end{cases}$$

$\equiv$ { introdução de variáveis-+, def-comp }

$$\begin{cases} outExpAr\ \underline{X}\ a = i_1\ (a) \\ outExpAr\ (N\ a) = i_2\ (i_1\ (a)) \\ outExpAr\ (bin\ a) = i_2\ (i_2\ (i_1\ (a))) \\ outExpAr\ \widehat{Un}\ a = i_2\ (i_2\ (i_2\ (a))) \end{cases}$$

$\square$

Transformando esse sistema de equações para notação de **Haskell**, temos a solução:

$outExpAr :: ExpAr\ a \to () + (a + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a)))$
$outExpAr\ (X) = i_1\ ()$
$outExpAr\ (N\ a) = i_2\ (i_1\ (a))$
$outExpAr\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a)))$
$outExpAr\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b))))$

recExpAr

Sabendo agora o *tipo de saída do outExpAr*, passamos a conhecer o *tipo de entrada da função recExpAr*. Como tal função será a responsável por chamar recursivamente o catamorfismo para as *"ExpAr's"*presentes no tipo de entrada, basta separarmos a função nos casos específicos em que temos de invocar o catamorfismo recursivamente, isto é, caso o tipo de entrada for do tipo **(BinOp,(ExpAr a,ExpAr a)** ou **(UnOp,ExpAr a**. Caso contrário, não haverá recursividade e basta invocarmos o **id**

Para uma melhor ilustração deste processo, segue um diagrama do estado da resolução até este momento:

$$\begin{tikzcd} ExpAr\ a \arrow[r, "outExpAr"'] \arrow[d, "cataExpAr\ (gene)"'] & 1 + (A + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a))) \arrow[d, "recExpAr\ (cataExpAr\ (gene))"] \\ (Floating)\ C & 1 + (A + ((BinOp, (C, C) + (UnOp, C)))) \arrow[l, "gene"] \end{tikzcd}$$

Concluindo, temos então recExpAr como:

$recExpAr\ f = id + (id + ((f1\ f) + (f2\ f)))$ **where**
$f1\ f\ (op, (a, b)) = (op, (f\ a, f\ b))$
$f2\ f\ (op, a) = (op, f\ a)$

g$_e val_e xp$

$g\_eval\_exp\ a = [\underline{a}, resto]$ **where**
$resto = [id, resto2]$
$resto2 = [bin, un]$ **where**
$bin\ (Sum, (c, d)) = c + d$
$bin\ (Product, (c, d)) = c * d$
$un\ (Negate, c) = (-1) * c$
$un\ (E, c) = expd\ c$

clean

$$
\begin{array}{l}
clean :: (Floating\ a, Eq\ a) \Rightarrow ExpAr\ a \rightarrow () + (a + ((BinOp, (ExpAr\ a, ExpAr\ a)) + (UnOp, ExpAr\ a)))\\
clean\ (X) = i_1\ ()\\
clean\ (N\ a) = i_2\ (i_1\ (a))\\
clean\ (Un\ op\ a) \mid (op \equiv E) \wedge a \equiv (N\ 0) = i_2\ (i_1\ (1))\\
\quad \mid otherwise = i_2\ (i_2\ (i_2\ (op, a)))\\
clean\ (Bin\ op\ a\ b) \mid (op \equiv Product) \wedge (a \equiv (N\ 0) \vee b \equiv (N\ 0)) = i_2\ (i_1\ (0))\\
\qquad\qquad\qquad\qquad \mid otherwise = i_2\ (i_2\ (i_1\ (op, (a, b))))
\end{array}
$$

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

sd_gen

$$
\begin{array}{l}
sd\_gen :: Floating\ a \Rightarrow\\
\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \rightarrow (ExpAr\ a\\
sd\_gen = [\langle \underline{X}, N \cdot \underline{1} \rangle, resto]\ \textbf{where}\\
\quad resto = [\langle N \cdot id, N \cdot \underline{0} \rangle, resto2]\ \textbf{where}\\
\quad resto2 = [f1, f2]\ \textbf{where}\\
\quad f1\ (Sum, ((a, b), (c, d))) = (Bin\ Sum\ a\ c, Bin\ Sum\ b\ d)\\
\quad f1\ (Product, ((a, b), (c, d))) = (Bin\ Product\ a\ c, Bin\ Sum\ (Bin\ Product\ a\ d)\ (Bin\ Product\ b\ c))\\
\quad f2\ (Negate, (a, b)) = (Un\ Negate\ a, Un\ Negate\ b)\\
\quad f2\ (E, (a, b)) = (Un\ E\ a, Bin\ Product\ (Un\ E\ a)\ b)\\
\qquad \text{-- (a,b) originais (c,b) derivadas}
\end{array}
$$

ad_gen

$$
\begin{array}{l}
ad\_gen :: Floating\ a \Rightarrow\\
\quad a \rightarrow () + (a + ((BinOp, ((ExpAr\ a, a), (ExpAr\ a, a))) + (UnOp, (ExpAr\ a, a)))) \rightarrow (ExpAr\ a, a)\\
ad\_gen\ v = [\langle \underline{X}, \underline{1} \rangle, resto]\ \textbf{where}\\
\quad resto = [\langle N \cdot id, \underline{0} \rangle, resto2]\ \textbf{where}\\
\quad resto2 = [f1, f2]\ \textbf{where}\\
\quad f1\ (Sum, ((a, b), (c, d))) = (Bin\ Sum\ a\ c, b + d)\\
\quad f1\ (Product, ((a, b), (c, d))) = (Bin\ Product\ a\ c, ((eval\_exp\ v\ a) * d) + (b * (eval\_exp\ v\ c)))\\
\quad f2\ (Negate, (a, b)) = (Un\ Negate\ a, (-1) * b)\\
\quad f2\ (E, (a, b)) = (Un\ E\ a, (expd\ (eval\_exp\ v\ a)) * b)
\end{array}
$$

Problema 2

Definir

$$
\begin{array}{l}
loop\ (top, bot, n) = (2 * n * (2 * n - 1) * top, n * n * bot, 1 + n)\\
inic = (1, 1, 1)\\
prj\ (top, bot, n) = (top\ `div`\ (n * bot))
\end{array}
$$

por forma a que

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

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

Começamos por separar o cálculo em duas partes, a parte do numerador e a do denominador. Se nos forcamos no denominador, $(n+1)!(n!)$, conseguimos determinar matematicamente uma forma de o calcular:

$(n+1)!(n!) = (n+1)(n!)(n!) = (n+1)(n!)^2$

Conseguimos, então, criar uma função recursiva que calcula $(n!)^2$:

$$
\begin{aligned}
&bot\ 0 = 1\\
&bot\ (n+1) = (bot\ n) * n * n
\end{aligned}
$$

Podemos fazer o mesmo para o numerador, $(2n)!$:
$(2n)! = (2n) * (2n-1) * (2(n-1))!$

$$
\begin{aligned}
&top\ 0 = 1\\
&top\ (n+1) = (2 * n * (2 * n - 1)) * (top\ n)
\end{aligned}
$$

Utilizando isto, conseguimos determinar então uma função final que calcula o valor final:

$$cat\ n = (top\ n)\ `div`\ ((n+1) * (bot\ n))$$

Podemos definir todas as funções necessárias para utilizar a regra de algibeira:

$$
\begin{aligned}
&bot\ 0 = 1\\
&bot\ (n+1) = (bot\ n) * (s\ n) * (s\ n)\\
&top\ 0 = 1\\
&top\ (n+1) = (2 * (s\ n) * (2 * (s\ n) - 1)) * (top\ n)\\
&s\ 0 = 1\\
&s\ (n+1) = 1 + s\ n
\end{aligned}
$$

Aplicando a regra de algibeira chegamos então à solução apresentada:

$$
\begin{aligned}
&cat = prj \cdot \mathsf{for}\ loop\ inic\ \mathbf{where}\\
&\quad loop\ (top, bot, n) = (2 * n * (2 * n - 1) * top, n * n * bot, 1 + n)\\
&\quad inic = (1, 1, 1)\\
&\quad prj\ (top, bot, n) = (top\ `div`\ (n * bot))
\end{aligned}
$$

Problema 3

$$
\begin{aligned}
&\left\{\begin{array}{l} calcLine\ [\,] = \underline{nil}\\ calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x)\end{array}\right.\\
\equiv\quad&\{\ \text{nil x = [] }\}\\
&\left\{\begin{array}{l} calcLine\ (nil\ x) = \underline{nil}\\ calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x)\end{array}\right.\\
\equiv\quad&\{\ \text{Def-comp }\}\\
&\left\{\begin{array}{l} (calcLine \cdot nil)\ x = \underline{nil}\\ calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x)\end{array}\right.\\
\equiv\quad&\{\ \text{Def-const }\}\\
&\left\{\begin{array}{l} (calcLine \cdot nil)\ x = \underline{\underline{nil}}\ x\\ calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x)\end{array}\right.\\
\equiv\quad&\{\ \text{nil x = []; def-comp, def-const, cons(h,t) = h:t }\}\\
&\left\{\begin{array}{l} (calcLine \cdot nil)\ x = \underline{\underline{nil}}\ x\\ (calcLine \cdot cons)\ (p, x) = \overline{g}\ p\ (calcLine\ x)\end{array}\right.\\
\equiv\quad&\{\ \text{def curry, Igualdade extensional }\}\\
&\left\{\begin{array}{l} calcLine \cdot nil = \underline{\underline{nil}}\\ (calcLine \cdot cons)\ (p, x) = g\ (p, calcLine\ x)\end{array}\right.\\
\equiv\quad&\{\ \text{Def-x, def-id, igualdade extensional, def-comp }\}\\
&\left\{\begin{array}{l} calcLine \cdot nil = \underline{\underline{nil}}\\ calcLine \cdot cons = \overline{f} \cdot (id\ x\ calcLine)\end{array}\right.
\end{aligned}
$$

$$
\begin{array}{ll}
\equiv & \{\ \text{Eq-+}\ \} \\
& [\,calcLine \cdot nil, calcLine \cdot cons\,] = [\,\underline{\underline{nil}}, g \cdot (id \times calcLine)\,] \\
\equiv & \{\ \text{Fusão-+}\ \} \\
& calcLine \cdot [\,nil, cons\,] = [\,\underline{\underline{nil}}, g \cdot (id \times calcLine)\,] \\
\equiv & \{\ \text{Nat-id, absorção-+}\ \} \\
& calcLine \cdot [\,nil, cons\,] = [\,\underline{\underline{nil}}, g\,] \cdot (id + id \times calcLine) \\
\equiv & \{\ \text{inL = [nil,cons], F f = id + id x f}\ \} \\
& calcLine \cdot [\,nil, cons\,] = [\,\underline{\underline{nil}}, g\,] \cdot F\ calcLine \\
\equiv & \{\ \text{Universal-cata}\ \} \\
& calcLine = (\!|\, [\,\underline{\underline{nil}}, g\,] \,|\!) \\
\Box &
\end{array}
$$

calcLine :: NPoint -¿ (NPoint -¿ OverTime NPoint) calcLine = cataList (either (const (const nil)) g) where g :: (Rational, NPoint -¿ OverTime NPoint) -¿ (NPoint -¿ OverTime NPoint) g (d,f) l = case l of [] -¿ nil (x:xs) -¿ -¿ concat $(sequenceA[singl.linear1ddx, fxs])z$

$$
\begin{array}{l}
deCasteljau :: [\,NPoint\,] \rightarrow OverTime\ NPoint \\
deCasteljau = hyloAlgForm\ alg\ coalg\ \textbf{where} \\
\quad coalg = divide \\
\quad alg = conquer \\
divide\ [\,] = i_1\ [\,] \\
divide\ [\,a\,] = i_1\ a \\
divide\ l = i_2\ (init\ l, tail\ l) \\
quer\ (x, y) = \lambda pt \rightarrow calcLine\ (x\ pt)\ (y\ pt)\ pt \\
conquer = [\underline{\cdot}, quer] \\
hyloAlgForm\ f\ g = (\!|\, f \,|\!) \cdot [\!(\, g \,)\!]
\end{array}
$$

Problema 4

Solução para listas não vazias:

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

$$
\begin{tikzcd}
A^+ \arrow[d, "\langle avg, length \rangle"'] & & A + A \times A^+ \arrow[ll, "\mathsf{in}"'] \arrow[d, "id + \langle avg, length \rangle"] \\
A \times \mathbb{N}_0 & & A + A \times (A \times \mathbb{N}_0) \arrow[ll, "g"]
\end{tikzcd}
$$

A partir do diagrama facilmente chegamos ao gene. Caso a sequência seja unitária então o resultado é o próprio número com length de 1. Se não, aplicamos a definição de avg e aplicamos a função succ à length que já tinha sido calculada.

$$
\begin{array}{l}
outSList\ ([\,a\,]) = i_1\ (a) \\
outSList\ (a : b) = i_2\ (a, b) \\
cataSList\ g = g \cdot (id + id \times cataSList\ g) \cdot outSList \\
avg\_aux = cataSList\ [\langle id, one \rangle, \langle k, \mathsf{succ}\ \cdot \pi_2 \cdot \pi_2 \rangle] \\
\quad \textbf{where} \\
\qquad k :: (Double, (Double, Integer)) \rightarrow Double \\
\qquad k\ (a, (b, c)) = (a + c' * b)\ /\ (c' + 1)
\end{array}
$$

$$\begin{aligned}
&\quad \textbf{where} \\
&\quad\quad c' = \mathit{fromIntegral}\ c
\end{aligned}$$

Solução para árvores de tipo LTree:

$$\begin{aligned}
&\mathit{avgLTree} = \pi_1 \cdot (\!|\, \mathit{gene} \,|\!) \ \textbf{where} \\
&\quad \mathit{gene} = [\langle \mathit{id}, \mathit{one} \rangle, \langle k, \mathit{add} \cdot (\pi_2 \times \pi_2) \rangle] \\
&\quad k :: ((\mathit{Double}, \mathit{Integer}), (\mathit{Double}, \mathit{Integer})) \to \mathit{Double} \\
&\quad k\ ((a, b), (c, d)) = (a * b' + c * d')\ /\ (b' + d') \\
&\quad\quad \textbf{where} \\
&\quad\quad\quad b' = \mathit{fromIntegral}\ b \\
&\quad\quad\quad d' = \mathit{fromIntegral}\ d
\end{aligned}$$

Problema 5

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

F# mostrou ser uma linguagem bastante semelhante a Haskell, sendo a diferença principal a sintaxe (por exemplo, requirir o uso de *let* antes da definição de funções). A parte mais difícil desta ”tradução”foi a diferença na omissão de argumentos, algo que, em geral, não é possível em F# .

```
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 x = (g << recBTree (cataBTree g) << outBTree) x
let rec anaBTree g x = (inBTree << recBTree (anaBTree g) << g) x
let hyloBTree h g x = (cataBTree h << anaBTree g) x


// (3) Map ---------------------------------------------------------------------
let fmap f x = cataBTree (inBTree << baseBTree f id) x

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

// (4.1) Inmersion (mirror) ----------------------------------------------------
let invBTree x = cataBTree (inBTree << (id -|- (id >< swap))) x

// (4.2) Counting --------------------------------------------------------------
let countBTree x = cataBTree (either (konst 0) (succ << (uncurry (+)) << p2)) x

// (4.3) Serialization ---------------------------------------------------------
let inord y =
    let join (x, (l, r)) = l @ [x] @ r
    either nil join y
let inordt x = cataBTree inord x // in-order traversal

let preord y =
```

```
    let f (x, (l, r)) = x::(l @ r)
    either nil f y
let preordt x = cataBTree preord x // pre-order traversal

let postordt y = // post-order traversal
    let f (x, (l, r)) = l @ r @ [x]
    cataBTree (either nil f) y

// (4.4) Quicksort ---------------------------------------------------------------
let rec part p x =
    match x with
    | [] -> ([], [])
    | (h::t) -> let s, l = part p t
                if p h then (h::s,l) else (s,h::l)

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

let qSort x = hyloBTree inord qsep x

// (4.5) Traces ------------------------------------------------------------------
let rec union a b =
    match b with
    | [] -> a
    | h::t when List.contains h a -> union a t
    | h::t -> h::(union a t)

let tunion (a, (l, r)) = union (List.map (fun x -> a::x) l) (List.map (fun x -> a::x)
let traces x = cataBTree (either (konst [[]]) tunion) x

// (4.6) Towers of Hanoi -------------------------------------------------------
let present = inord
let strategy (d, n) = if n = 0 then i1 () else i2 ((n - 1, d), ((not d, n - 1), (not

let hanoi x = hyloBTree present strategy x

// The Towers of Hanoi problem comes from a puzzle marketed in 1883
// by the French mathematician Édouard Lucas, under the pseudonym
// Claus. The puzzle is based on a legend according to which
// there is a temple, apparently in Bramah rather than in Hanoi as
// one might expect, where there are three giant poles fixed in the
// ground. On the first of these poles, at the time of the world’s
// creation, God placed sixty four golden disks, each of different
// size, in decreasing order of size. The Bramin monks were given
// the task of moving the disks, one per day, from one pole to another
// subject to the rule that no disk may ever be above a smaller disk.
// The monks’ task would be complete when they had succeeded in moving
// all the disks from the first of the poles to the second and, on
// the day that they completed their task the world would come to
// an end!

// There is a wellknown inductive solution to the problem given
// by the pseudocode below. In this solution we make use of the fact
// that the given problem is symmetrical with respect to all three
// poles. Thus it is undesirable to name the individual poles. Instead
```

```
// we visualize the poles as being arranged in a circle; the problem
// is to move the tower of disks from one pole to the next pole in
// a specified direction around the circle. The code defines H n d
// to be a sequence of pairs (k,d’) where n is the number of disks,
// k is a disk number and d and d’ are directions. Disks are numbered
// from 0 onwards, disk 0 being the smallest. (Assigning number 0
// to the smallest rather than the largest disk has the advantage
// that the number of the disk that is moved on any day is independent
// of the total number of disks to be moved.) Directions are boolean
// values, true representing a clockwise movement and false an anticlockwise
// movement. The pair (k,d’) means move the disk numbered k from
// its current position in the direction d’. The semicolon operator
// concatenates sequences together, [] denotes an empty sequence
// and [x] is a sequence with exactly one element x. Taking the pairs
// in order from left to right, the complete sequence H n d prescribes
// how to move the n smallest disks onebyone from one pole to the
// next pole in the direction d following the rule of never placing
// a larger disk on top of a smaller disk.

// H 0     d = [],
// H (n+1) d = H n d ; [ (n, d) ] ; H n d.

// (excerpt from R. Backhouse, M. Fokkinga / Information Processing
// Letters 77 (2001) 71--76)


// (5) Depth and balancing (using mutual recursion) --------------------------
let baldepth n =
    let h (a, ((b1, b2), (d1, d2))) = (b1 && b2 && abs (d1 - d2) <= 1, 1 + max d1 d2
    let f ((b1, d1), (b2, d2)) = ((b1, b2), (d1, d2))
    let g x = either (konst (true, 1)) (h << (id >< f)) x
    cataBTree g n

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