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

Departamento de Informática
Universidade do Minho

Maio de 2022

Grupo nr.	119
a89492	Ivo Pereira Vilas Boas
a89509	Carlos Humberto da Silva Ferreira
a89555	Diogo Francisco Lima Barbosa
a90439	Nuno Gonçalo Machado Rodrigues

1 Preâmbulo

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

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

2 Documentação

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

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

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

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

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

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

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

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

```
$ ghci cp2021t.lhs
```

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

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

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

é seleccionado pelo GHCi para ser executado.

3 Como realizar o trabalho

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

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

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

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

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

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

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

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

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

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

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

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

3.1 Stack

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

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

Pode aceder ao GHCi utilizando o comando:

```
stack ghci
```

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

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

Problema 1

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

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

- *Symbolic differentiation*
- *Automatic differentiation*

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

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

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

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

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

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

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 13 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 13 expressa como um hilomorfismo[2] e teste as propriedades:

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

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

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

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

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

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

- Regra do produto:

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

Defina o gene do catamorfismo que ocorre na função

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

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

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

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

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

   Defina o gene do catamorfismo que ocorre na função

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

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

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

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

Problema 2

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

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

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

[4] Lei (3.94) em [2], página 98.

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

Obter-se-á de imediato

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

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:

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

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

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

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

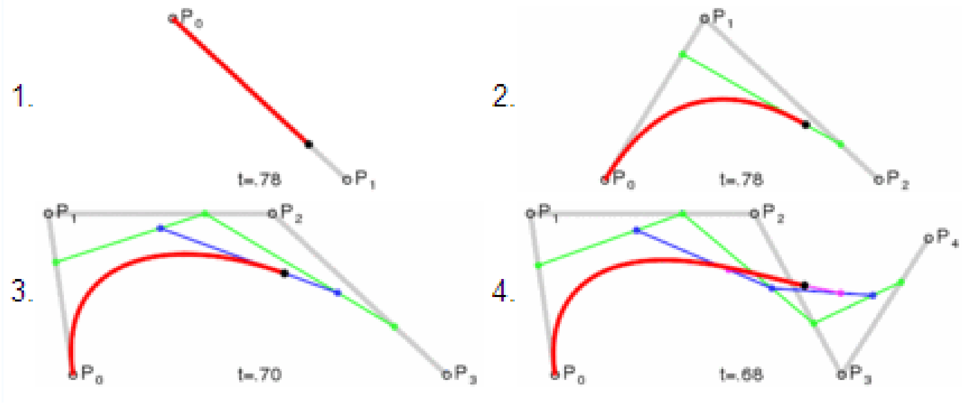

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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

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

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

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

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

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

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 $Float$).

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

O anexo C tem definida a função

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

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

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

que implementa o algoritmo respectivo.

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

   Propriedade [QuickCheck] 8 *Definição alternativa.*

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

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

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

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

3. Corra a função $\mathit{runBezier}$ 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,

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

onde $k = \mathit{length}\ x$. 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

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

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 $\mathit{avg\_aux} = (\![[b, q]]\!)$ tal que $\mathit{avg\_aux} = \langle \mathit{avg}, \mathit{length}\rangle$ 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:*

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

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 [2].
[9] Cf. [2], página 102.

C Código fornecido

Problema 1

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

Problema 2

Definição da série de Catalan usando factoriais (1):

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

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

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

Problema 3

Algoritmo:

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

Função auxiliar:

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

2D:

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

Modelo:

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

[10] Fonte: Wikipedia.

$$tick :: Float \rightarrow World \rightarrow World$$
$$tick\ dt\ world = world\ \{time = (time\ world) + dt\}$$

$$actions :: Event \rightarrow World \rightarrow World$$
$$actions\ (EventKey\ (MouseButton\ LeftButton)\ Down\ \_\ p)\ world =$$
$$\quad world\ \{points = (points\ world) +\!\!+ [(\lambda(x, y) \rightarrow \mathsf{map}\ \ to_{\mathbb{Q}}\ [x, y])\ p]\}$$
$$actions\ (EventKey\ (SpecialKey\ KeyDelete)\ Down\ \_\ \_)\ world =$$
$$\quad world\ \{points = cond\ (\equiv [\,])\ id\ init\ (points\ world)\}$$
$$actions\ \_\ world = world$$

$$scaleTime :: World \rightarrow Float$$
$$scaleTime\ w = (1 + cos\ (time\ w))\ /\ 2$$

$$bezier2dAtTime :: World \rightarrow (Float, Float)$$
$$bezier2dAtTime\ w = (bezier2dAt\ w)\ (scaleTime\ w)$$

$$bezier2dAt :: World \rightarrow OverTime\ (Float, Float)$$
$$bezier2dAt\ w = bezier2d\ (points\ w)$$

$$thicCirc :: Picture$$
$$thicCirc = ThickCircle\ 4\ 10$$

$$ps :: [Float]$$
$$ps = \mathsf{map}\ \ from_{\mathbb{Q}}\ ps'\ \mathbf{where}$$
$$\quad ps' :: [\mathbb{Q}]$$
$$\quad ps' = [0, 0.01\,..\,1] \quad \text{-- interval}$$

Gloss:

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

Animação:

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

Propriedades e *main*:

$$runBezier :: \mathsf{IO}\ ()$$
$$runBezier = play\ (InWindow\ \texttt{"Bézier"}\ (600, 600)\ (0, 0))$$
$$\quad black\ 50\ initW\ picture\ actions\ tick$$

$$runBezierSym :: \mathsf{IO}\ ()$$
$$runBezierSym = quickCheckWith\ (stdArgs\ \{maxSize = 20, maxSuccess = 200\})\ prop\_bezier\_sym$$

Compilação e execução dentro do interpretador:[11]

$$main = runBezier$$

$$run = \mathbf{do}\ \{system\ \texttt{"ghc cp2021t"}; system\ \texttt{"./cp2021t"}\}$$

[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:

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

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

Outras funções auxiliares

Lógicas:

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

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.

Funções dos alunos

Redefinição polinomial de funções no capítulo 8 da biblioteca Cp, *Basic functions, abbreviations*, que sofreram da **Restrição de Monomorsfismos**[12]:

> **"**
>
> *The "monomorphism restriction" is a counter-intuitive rule in Haskell type inference. If you forget to provide a type signature, sometimes this rule will fill the free type variables with specific types using "type defaulting" rules. The resulting type signature is always less polymorphic than you'd expect, so often this results in the compiler throwing type errors at you in situations where you expected it to infer a perfectly sane type for a polymorphic expression.*
>
> **"**

$$zeroP :: Num\ a \Rightarrow b \rightarrow a$$
$$zeroP = \underline{0}$$
$$oneP :: Num\ a \Rightarrow b \rightarrow a$$
$$oneP = \underline{1}$$
$$addP :: Num\ c \Rightarrow (c, c) \rightarrow c$$
$$addP = \widehat{(+)}$$
$$mulP :: Num\ c \Rightarrow (c, c) \rightarrow c$$
$$mulP = \widehat{(*)}$$

Problema 1

São dadas:

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

[12] Fonte: HaskellWiki.

Definir:

$$
\begin{array}{l}
\mathsf{out}_{ExpAr}\ X \qquad\qquad = i_1\ () \\
\mathsf{out}_{ExpAr}\ (N\ a) \qquad\quad = i_2\ \$\ i_1\ a \\
\mathsf{out}_{ExpAr}\ (Bin\ op\ a\ b) = i_2\ \$\ i_2\ \$\ i_1\ (op, (a, b)) \\
\mathsf{out}_{ExpAr}\ (Un\ op\ a) \quad = i_2\ \$\ i_2\ \$\ i_2\ (op, a) \\
\quad \text{--} \\
\mathsf{F}_{ExpAr}\ f = \mathsf{B}_{ExpAr}\ id\ id\ id\ f\ f\ id\ f \\
\quad \text{--} \\
g\_eval\_exp\ v = [var, [num, [bin, un]]]\ \mathbf{where} \\
\quad var\ \ () \qquad\qquad\quad = v \\
\quad num\ a \qquad\qquad\quad = a \\
\quad bin\ \ (Sum, (a, b)) \qquad = a + b \\
\quad bin\ \ (Product, (a, b)) = a * b \\
\quad un\ \ \ (Negate, a) \qquad = negate\ a \\
\quad un\ \ \ (E, a) \qquad\qquad = expd\ a \\
\quad \text{--} \\
eval\_exp\_int\ v = cataExpAr\ \$\ g\_eval\_exp\_int\ v \\
g\_eval\_exp\_int\ v = [var, [num, [bin, un]]]\ \mathbf{where} \\
\quad var\ \ () \qquad\qquad\quad = v \\
\quad num\ a \qquad\qquad\quad = a \\
\quad bin\ \ (Sum, (a, b)) \qquad = a + b \\
\quad bin\ \ (Product, (0, b)) = 0 \\
\quad bin\ \ (Product, (a, 0)) = 0 \\
\quad bin\ \ (Product, (a, b)) = a * b \\
\quad un\ \ \ (Negate, a) \qquad = negate\ a \\
\quad un\ \ \ (E, 0) \qquad\qquad = 1 \\
\quad un\ \ \ (E, a) \qquad\qquad = expd\ a \\
\quad \text{--} \\
clean\ (Bin\ Product\ (N\ 0)\ \_) = i_2\ \$\ i_1\ 0 \\
clean\ (Bin\ Product\ \_\ (N\ 0)) = i_2\ \$\ i_1\ 0 \\
clean\ (Un\ E\ (N\ 0)) \qquad\quad = i_2\ \$\ i_1\ 1 \\
clean\ a = \mathsf{out}_{ExpAr}\ a \\
\quad \text{--} \\
gopt\ a = g\_eval\_exp\ a
\end{array}
$$

$$
\begin{array}{l}
sd\_gen :: Floating\ a \Rightarrow \\
\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a)))) \\
\quad \to (ExpAr\ a, ExpAr\ a) \\
sd\_gen = [var, [num, [bin, un]]]\ \mathbf{where} \\
\quad var\ \_\ \ = (X, N\ 1) \\
\quad num\ a = (N\ a, N\ 0) \\
\quad bin\ (Sum \qquad, ((x, x'), (y, y'))) = (Bin\ Sum \qquad x\ y, Bin\ Sum\ x'\ y') \\
\quad bin\ (Product, ((x, x'), (y, y'))) = (Bin\ Product\ x\ y, Bin\ Sum\ (Bin\ Product\ x\ y')\ (Bin\ Product\ x'\ y)) \\
\quad un\ \ (Negate\ \ , (x, x')) \qquad\qquad = (Un\ \ Negate\ \ x \quad, Un\ \ Negate\ x') \\
\quad un\ \ (E \qquad\quad , (x, x')) \qquad\qquad = (Un\ \ E \qquad\quad x \quad, Bin\ Product\ (Un\ E\ x)\ x')
\end{array}
$$

$$
\begin{array}{l}
ad\_gen\ v = [var, [num, [bin, un]]]\ \mathbf{where} \\
\quad var\ \_ = (v, 1) \\
\quad num\ a = (a, 0) \\
\quad bin\ (Sum \qquad, ((x, x'), (y, y'))) = (x + y \quad\ , x' + y') \\
\quad bin\ (Product, ((x, x'), (y, y'))) = (x * y \quad\ \ , x * y' + x' * y) \\
\quad un\ \ (Negate\ \ , (x, x')) \qquad\qquad = (negate\ x, negate\ x') \\
\quad un\ \ (E \qquad\quad , (x, x')) \qquad\qquad = (expd\ x \quad, expd\ x * x')
\end{array}
$$

Prova da definição de outExpAr

$$
\begin{tikzcd}
ExpAr\ A \arrow[rr, bend left, "\mathsf{out}_{ExpAr}"] \arrow[rr, phantom, "\cong"] & & OutExpAr\ A \arrow[ll, bend left, "inExpAr"]
\end{tikzcd}
$$

$$
\begin{aligned}
& \mathsf{out}_{ExpAr} \cdot inExpAr = id \\
\equiv \quad & \{ \text{ Def-inExpAr } \} \\
& \mathsf{out}_{ExpAr} \cdot [\underline{X}, [N, [bin, \widehat{Un}]]] = id \\
\equiv \quad & \{ \text{ Fusão-+ } \} \\
& [\mathsf{out}_{ExpAr} \cdot \underline{X}, \mathsf{out}_{ExpAr} \cdot [N, [bin, \widehat{Un}]]] = id \\
\equiv \quad & \{ \text{ Universal-+; Natural-id } \} \\
& \begin{cases} \mathsf{out}_{ExpAr} \cdot \underline{X} = i_1 \\ \mathsf{out}_{ExpAr} \cdot [N, [bin, \widehat{Un}]] = i_2 \end{cases} \\
\equiv \quad & \{ \text{ Fusão-+ } \} \\
& \begin{cases} \mathsf{out}_{ExpAr} \cdot \underline{X} = i_1 \\ [\mathsf{out}_{ExpAr} \cdot N, \mathsf{out}_{ExpAr} \cdot [bin, \widehat{Un}]] = i_2 \end{cases} \\
\equiv \quad & \{ \text{ Universal-+; Natural-id } \} \\
& \begin{cases} \mathsf{out}_{ExpAr} \cdot \underline{X} = i_1 \\ \begin{cases} \mathsf{out}_{ExpAr} \cdot N = i_2 \cdot i_1 \\ \mathsf{out}_{ExpAr} \cdot [bin, \widehat{Un}] = i_2 \cdot i_2 \end{cases} \end{cases} \\
\equiv \quad & \{ \text{ Fusão-+ } \} \\
& \begin{cases} \mathsf{out}_{ExpAr} \cdot \underline{X} = i_1 \\ \begin{cases} \mathsf{out}_{ExpAr} \cdot N = i_2 \cdot i_1 \\ [\mathsf{out}_{ExpAr} \cdot bin, \mathsf{out}_{ExpAr} \cdot \widehat{Un}] = i_2 \cdot i_2 \end{cases} \end{cases} \\
\equiv \quad & \{ \text{ Universal-+; Natural-id } \} \\
& \begin{cases} \mathsf{out}_{ExpAr} \cdot \underline{X} = i_1 \\ \begin{cases} \mathsf{out}_{ExpAr} \cdot N = i_2 \cdot i_1 \\ \begin{cases} \mathsf{out}_{ExpAr} \cdot bin = i_2 \cdot i_2 \cdot i_1 \\ \mathsf{out}_{ExpAr} \cdot \widehat{Un} = i_2 \cdot i_2 \cdot i_2 \end{cases} \end{cases} \end{cases} \\
\equiv \quad & \{ \text{ Igualdade extensional; Def-comp } \} \\
& \begin{cases} \mathsf{out}_{ExpAr}\ (\underline{X}\ ()) = i_1\ () \\ \begin{cases} \mathsf{out}_{ExpAr}\ (N\ a) = i_2\ (i_1\ a) \\ \begin{cases} \mathsf{out}_{ExpAr}\ (bin\ (op, (a, b))) = i_2\ (i_2\ (i_1\ (op, (a, b)))) \\ \mathsf{out}_{ExpAr}\ (\widehat{Un}\ (op, a)) = i_2\ (i_2\ (i_2\ (op, a))) \end{cases} \end{cases} \end{cases} \\
\equiv \quad & \{ \text{ Def-const; Def-bin; Def-Un } \} \\
& \begin{cases} \mathsf{out}_{ExpAr}\ X = i_1\ () \\ \begin{cases} \mathsf{out}_{ExpAr}\ (N\ a) = i_2\ (i_1\ a) \\ \begin{cases} \mathsf{out}_{ExpAr}\ (Bin\ op\ a\ b) = i_2\ (i_2\ (i_1\ (op, (a, b)))) \\ \mathsf{out}_{ExpAr}\ (Un\ op\ a) = i_2\ (i_2\ (i_2\ (op, a))) \end{cases} \end{cases} \end{cases}
\end{aligned}
$$

$\square$

Prova da definição de g_eval_exp

$$\begin{tikzcd}
ExpAr\ A \arrow[d, "eval_exp\ v"'] & & 1 + (A + (BinOp \times (ExpAr\ A)^2 + UnOp \times ExpAr\ A)) \arrow[ll, "inExpAr"'] \arrow[d, "id+(id+(id\times(eval_exp\ v\times eval_exp\ v)+id\times eval_exp\ v))"] \\
A & & 1 + (A + (BinOp \times A^2 + UnOp \times A)) \arrow[ll, "g_eval_exp\ v"]
\end{tikzcd}$$

$$(eval\_exp\ v) \cdot inExpAr = g\_eval\_exp \cdot \mathsf{F}\ (eval\_exp\ v)$$

$\equiv$ { Def-inExpAr; Def-F; ev v := eval_exp v; g v := g_eval_exp v }

$$(ev\ v) \cdot [\underline{X}, [N, [bin, \widehat{Un}]]] = g \cdot (id + (id + (id \times (ev\ v \times ev\ v) + id \times ev\ v)))$$

$\equiv$ { Inferência do tipo de g }

$$(ev\ v) \cdot [\underline{X}, [N, [bin, \widehat{Un}]]] = [g_1, [g_2, [g_3, g_4]]] \cdot (id + (id + (id \times (ev\ v \times ev\ v) + id \times ev\ v)))$$

$\equiv$ { $3 \times$ Fusão-+; $3 \times$ Asborsção-+; $2 \times$ Natural-id }

$$[(ev\ v) \cdot \underline{X}, [(ev\ v) \cdot N, [(ev\ v) \cdot bin, (ev\ v) \cdot \widehat{Un}]]] = [g_1, [g_2, [g_3 \cdot (id \times (ev\ v \times ev\ v)), g_4 \cdot (id \times ev\ v)]]]$$

$\equiv$ { $3 \times$ Eq-+; f = g $\equiv$ g = f }

$$\begin{cases} g_1 = (ev\ v) \cdot \underline{X} \\ \begin{cases} g_2 = (ev\ v) \cdot N \\ \begin{cases} g_3 \cdot (id \times (ev\ v \times ev\ v)) = (ev\ v) \cdot bin \\ g_4 \cdot (id \times ev\ v) = (ev\ v) \cdot \widehat{Un} \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Igualdade extensional; Def-comp }

$$\begin{cases} g_1\ () = ev\ v\ (\underline{X}\ ()) \\ \begin{cases} g_2\ a = ev\ v\ (N\ a) \\ \begin{cases} g_3\ ((id \times (ev\ v \times ev\ v))\ (binop, (a, b))) = ev\ v\ (bin\ (binop, (a, b))) \\ g_4\ ((id \times ev\ v)\ (unop, a)) = ev\ v\ (\widehat{Un}\ (unop, a)) \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Def-const; Def-N; Def-bin; Def-$\widehat{Un}$; Def-$\times$ }

$$\begin{cases} g_1\ () = ev\ v\ X \\ \begin{cases} g_2\ a = ev\ v\ (N\ a) \\ \begin{cases} g_3\ (binop, (ev\ v\ a, ev\ v\ b)) = ev\ v\ (Bin\ binop\ a\ b) \\ g_4\ (unop, ev\ v\ a) = ev\ v\ (Un\ unop\ a) \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Pattern matching em binop e unop }

$$\begin{cases} g_1\ () = ev\ v\ X \\ \begin{cases} g_2\ a = ev\ v\ (N\ a) \\ \begin{cases} \begin{cases} g_3\ (Sum, (ev\ v\ a, ev\ v\ b)) = ev\ v\ (Bin\ Sum\ a\ b) \\ g_3\ (Product, (ev\ v\ a, ev\ v\ b)) = ev\ v\ (Bin\ Product\ a\ b) \end{cases} \\ \begin{cases} g_4\ (Negate, ev\ v\ a) = ev\ v\ (Un\ Negate\ a) \\ g_4\ (E, ev\ v\ a) = ev\ v\ (Un\ E\ a) \end{cases} \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Def-ev; *var* := g_1; *num* := g_2; *bin* := g_3; *un* := g_4 }

$$\begin{cases} var\ () = v \\ \begin{cases} num\ a = a \\ \begin{cases} \begin{cases} bin\ (Sum, (v1, v2)) = v1 + v2 \\ bin\ (Product, (v1, v2)) = v1 * v2 \end{cases} \\ \begin{cases} un\ (Negate, v1) = negate\ v1 \\ un\ (E, v1) = expd\ v1 \end{cases} \end{cases} \end{cases} \end{cases}$$

$\square$

Propriedades usadas para definição de clean

Elemento absovente da multiplicação:

$$\begin{aligned} x * 0 &= 0 \\ 0 * x &= 0 \end{aligned}$$

Propriedade Expoente Zero:

$$e^0 = 1$$

Provas das definições de sd_gen e ad_gen

Devido à necessiade de conhecer não só as derivadas dos subtermos do produto e da exponenciação, mas também os seus valores originais de forma a fazer a sua derivação usámos um **Paramorfismo**[13]. Algo que também é sugerido pelo *wrapper*, π_2, das funções sd e ad.

> **"**
>
> *In formal methods of computer science, a paramorphism (from Greek $\pi\alpha\rho\acute{\alpha}$, meaning "close together") is an extension of the concept of catamorphism first introduced by Lambert Meertens to deal with a form which "eats its argument and keeps it too".*
>
> *It is a more convenient version of catamorphism in that it gives the combining step function immediate access not only to the result value recursively computed from each recursive subobject, but the original subobject itself as well.*
>
> **"**

Diagramas dos catamorfismos presentes em sd e ad respetivamente:

$$\begin{tikzcd} ExpAr\ A \arrow[d, "\langle id, sd \rangle"'] & 1 + (A + (BinOp \times (ExpAr\ A)^2 + UnOp \times ExpAr\ A)) \arrow[l, "inExpAr"'] \arrow[d, "id + (id + (id \times (\langle id, sd \rangle \times \langle id, sd \rangle) + id \times \langle id, sd \rangle))"] \\ (ExpAr\ A)^2 & 1 + (A + (BinOp \times ((ExpAr\ A)^2 \times (ExpAr\ A)^2) + UnOp \times (ExpAr\ A)^2)) \arrow[l, "sd\_gen"] \end{tikzcd}$$

$$\begin{tikzcd} ExpAr\ A \arrow[d, "\langle id, ad\ v \rangle"'] & 1 + (A + (BinOp \times (ExpAr\ A)^2 + UnOp \times ExpAr\ A)) \arrow[l, "inExpAr"'] \arrow[d, "id + (id + (id \times (\langle id, ad\ v \rangle \times \langle id, ad\ v \rangle) + id \times \langle id, ad\ v \rangle))"] \\ A & 1 + (A + (BinOp \times (A^2 \times A^2) + UnOp \times A^2)) \arrow[l, "ad\_gen\ v"] \end{tikzcd}$$

[13] Fonte: Wikipedia.

Prova da definição de sd_gen

$$\langle id, sd \rangle \cdot inExpAr = sd\_gen \cdot \mathsf{F}\ (sd\_gen)$$

$\equiv$ { Def-inExpAr; Def-F }

$$\langle id, sd \rangle \cdot [\underline{X}, [N, [bin, \widehat{Un}]]] = sd\_gen \cdot (id + (id + (id \times \langle id, sd \rangle^2) + id \times \langle id, sd \rangle)))$$

$\equiv$ { Inferência do tipo de g }

$$\langle id, sd \rangle \cdot [\underline{X}, [N, [bin, \widehat{Un}]]] = [g_1, [g_2, [g_3, g_4]]] \cdot (id + (id + (id \times \langle id, sd \rangle^2) + id \times \langle id, sd \rangle)))$$

$\equiv$ { 3 × Fusão-+; 3 × Asborsção-+; 2 × Natural-id }

$$[\langle id, sd \rangle \cdot \underline{X}, [\langle id, sd \rangle \cdot N, [\langle id, sd \rangle \cdot bin, \langle id, sd \rangle \cdot \widehat{Un}]]] = [g_1, [g_2, [g_3 \cdot (id \times \langle id, sd \rangle^2)), g_4 \cdot (id \times \langle id, sd \rangle)]]]$$

$\equiv$ { 3 × Eq-+; f = g ≡ g = f }

$$\begin{cases} g_1 = \langle id, sd \rangle \cdot \underline{X} \\ \begin{cases} g_2 = \langle id, sd \rangle \cdot N \\ \begin{cases} g_3 \cdot (id \times \langle id, sd \rangle^2) = \langle id, sd \rangle \cdot bin \\ g_4 \cdot (id \times \langle id, sd \rangle) = \langle id, sd \rangle \cdot \widehat{Un} \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Igualdade extensional; Def-comp }

$$\begin{cases} g_1\ () = \langle id, sd \rangle\ (\underline{X}\ ()) \\ \begin{cases} g_2\ a = \langle id, sd \rangle\ (N\ a) \\ \begin{cases} g_3\ ((id \times \langle id, sd \rangle^2))\ (binop, (a, b))) = \langle id, sd \rangle\ (bin\ (binop, (a, b))) \\ g_4\ ((id \times \langle id, sd \rangle)\ (unop, a)) = \langle id, sd \rangle\ (\widehat{Un}\ (unop, a)) \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Def-const; Def-N; Def-bin; Def-$\widehat{Un}$; Def-× }

$$\begin{cases} g_1\ () = \langle id, sd \rangle\ X \\ \begin{cases} g_2\ a = \langle id, sd \rangle\ (N\ a) \\ \begin{cases} g_3\ (binop, (\langle id, sd \rangle\ a, \langle id, sd \rangle\ b)) = \langle id, sd \rangle\ (Bin\ binop\ a\ b) \\ g_4\ (unop, \langle id, sd \rangle\ a) = \langle id, sd \rangle\ (Un\ unop\ a) \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Def-split; Natural-id }

$$\begin{cases} g_1\ () = (X, sd\ X) \\ \begin{cases} g_2\ a = (N\ a, sd\ (N\ a)) \\ \begin{cases} g_3\ (binop, ((a, sd\ a), (b, sd\ b))) = (Bin\ binop\ a\ b, sd\ (Bin\ binop\ a\ b)) \\ g_4\ (unop, (a, sd\ a)) = (Un\ unop\ a, sd\ (Un\ unop\ a)) \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Pattern matching em binop e unop }

$$\begin{cases} g_1\ () = (X, sd\ X) \\ \begin{cases} g_2\ a = (N\ a, sd\ (N\ a)) \\ \begin{cases} \begin{cases} g_3\ (Sum, ((a, sd\ a), (b, sd\ b))) = (Bin\ Sum\ a\ b, sd\ (Bin\ Sum\ a\ b)) \\ g_3\ (Product, ((a, sd\ a), (b, sd\ b))) = (Bin\ Product\ a\ b, sd\ (Bin\ Product\ a\ b)) \end{cases} \\ \begin{cases} g_4\ (Negate, (a, sd\ a)) = (Un\ Negate\ a, sd\ (Un\ Negate\ a)) \\ g_4\ (E, (a, sd\ a)) = (Un\ E\ a, sd\ (Un\ E\ a)) \end{cases} \end{cases} \end{cases} \end{cases}$$

$\equiv$ { Def-sd; $var := g_1$; $num := g_2$; $bin := g_3$; $un := g_4$ }

$$\begin{cases} var\ () = (X, N\ 1) \\ \begin{cases} num\ a = (N\ a, N\ 0) \\ \begin{cases} \begin{cases} bin\ (Sum, ((a, a'), (b, b'))) = (Bin\ Sum\ a\ b, Bin\ Sum\ a'\ b') \\ bin\ (Product, ((a, a'), (b, b'))) = (Bin\ Product\ a\ b, Bin\ Product\ a'\ b') \end{cases} \\ \begin{cases} un\ (Negate, (a, a')) = (Un\ Negate\ a, Un\ Negate\ a') \\ un\ (E, (a, a')) = (Un\ E\ a, Un\ E\ a') \end{cases} \end{cases} \end{cases} \end{cases}$$

$\square$

Prova da definição de ad_gen

$$\langle id, ad\ v\rangle \cdot inExpAr = (ad\_gen\ v) \cdot \mathsf{F}\ (sd\_gen)$$

$\equiv$ $\{$ Def-inExpAr; Def-F $\}$

$$\langle id, ad\ v\rangle \cdot [\underline{X}, [N, [bin, \widehat{Un}]]] = (ad\_gen\ v) \cdot (id + (id + (id \times \langle id, ad\ v\rangle^2) + id \times \langle id, ad\ v\rangle)))$$

$\equiv$ $\{$ Inferência do tipo de g $\}$

$$\langle id, ad\ v\rangle \cdot [\underline{X}, [N, [bin, \widehat{Un}]]] = [g_1, [g_2, [g_3, g_4]]] \cdot (id + (id + (id \times \langle id, ad\ v\rangle^2) + id \times \langle id, ad\ v\rangle)))$$

$\equiv$ $\{$ $3 \times$ Fusão-+; $3 \times$ Asborsção-+; $2 \times$ Natural-id $\}$

$$[\langle id, ad\ v\rangle \cdot \underline{X}, [\langle id, ad\ v\rangle \cdot N, [\langle id, ad\ v\rangle \cdot bin, \langle id, ad\ v\rangle \cdot \widehat{Un}]]] = [g_1, [g_2, [g_3 \cdot (id \times \langle id, ad\ v\rangle^2)), g_4 \cdot (id \times$$

$\equiv$ $\{$ $3 \times$ Eq-+; f = g $\equiv$ g = f $\}$

$$\begin{cases} g_1 = \langle id, ad\ v\rangle \cdot \underline{X} \\ \begin{cases} g_2 = \langle id, ad\ v\rangle \cdot N \\ \begin{cases} g_3 \cdot (id \times \langle id, ad\ v\rangle^2) = \langle id, ad\ v\rangle \cdot bin \\ g_4 \cdot (id \times \langle id, ad\ v\rangle) = \langle id, ad\ v\rangle \cdot \widehat{Un} \end{cases} \end{cases} \end{cases}$$

$\equiv$ $\{$ Igualdade extensional; Def-comp $\}$

$$\begin{cases} g_1\ v = \langle id, ad\ v\rangle\ (\underline{X}\ v) \\ \begin{cases} g_2\ a = \langle id, ad\ v\rangle\ (N\ a) \\ \begin{cases} g_3\ ((id \times \langle id, ad\ v\rangle^2))\ (binop, (a, b))) = \langle id, ad\ v\rangle\ (bin\ (binop, (a, b))) \\ g_4\ ((id \times \langle id, ad\ v\rangle)\ (unop, a)) = \langle id, ad\ v\rangle\ (\widehat{Un}\ (unop, a)) \end{cases} \end{cases} \end{cases}$$

$\equiv$ $\{$ Def-const; Def-N; Def-bin; Def-$\widehat{Un}$; Def-$\times$ $\}$

$$\begin{cases} g_1\ v = \langle id, ad\ v\rangle\ X \\ \begin{cases} g_2\ a = \langle id, ad\ v\rangle\ (N\ a) \\ \begin{cases} g_3\ (binop, (\langle id, ad\ v\rangle\ a, \langle id, ad\ v\rangle\ b)) = \langle id, ad\ v\rangle\ (Bin\ binop\ a\ b) \\ g_4\ (unop, \langle id, ad\ v\rangle\ a) = \langle id, ad\ v\rangle\ (Un\ unop\ a) \end{cases} \end{cases} \end{cases}$$

$\equiv$ $\{$ Def-split; Natural-id $\}$

$$\begin{cases} g_1\ v = (X, (ad\ v)\ X) \\ \begin{cases} g_2\ a = (N\ a, (ad\ v)\ (N\ a)) \\ \begin{cases} g_3\ (binop, ((a, (ad\ v)\ a), (b, (ad\ v)\ b))) = (Bin\ binop\ a\ b, (ad\ v)\ (Bin\ binop\ a\ b)) \\ g_4\ (unop, (a, (ad\ v)\ a)) = (Un\ unop\ a, (ad\ v)\ (Un\ unop\ a)) \end{cases} \end{cases} \end{cases}$$

$\equiv$ $\{$ Pattern matching em binop e unop $\}$

$$\begin{cases} g_1\ v = (X, (ad\ v)\ X) \\ \begin{cases} g_2\ a = (N\ a, (ad\ v)\ (N\ a)) \\ \begin{cases} \begin{cases} g_3\ (Sum, ((a, (ad\ v)\ a), (b, (ad\ v)\ b))) = (Bin\ Sum\ a\ b, (ad\ v)\ (Bin\ Sum\ a\ b)) \\ g_3\ (Product, ((a, (ad\ v)\ a), (b, (ad\ v)\ b))) = (Bin\ Product\ a\ b, (ad\ v)\ (Bin\ Product\ a\ b)) \end{cases} \\ \begin{cases} g_4\ (Negate, (a, (ad\ v)\ a)) = (Un\ Negate\ a, (ad\ v)\ (Un\ Negate\ a)) \\ g_4\ (E, (a, (ad\ v)\ a)) = (Un\ E\ a, (ad\ v)\ (Un\ E\ a)) \end{cases} \end{cases} \end{cases} \end{cases}$$

$\equiv$ $\{$ Def-ad; $var := g_1$; $num := g_2$; $bin := g_3$; $un := g_4$ $\}$

$$\begin{cases} var\ v = (v, 1) \\ \begin{cases} num\ a = (a, 0) \\ \begin{cases} \begin{cases} bin\ (Sum, ((a, a'), (b, b'))) = (a + b, a' + b') \\ bin\ (Product, ((a, a'), (b, b'))) = (a * b, a * b' + a' * b) \end{cases} \\ \begin{cases} un\ (Negate, (a, a')) = (negate\ a, negate\ a') \\ un\ (E, (a, a')) = (expd\ a, expd\ a * a') \end{cases} \end{cases} \end{cases} \end{cases}$$

$\square$

Testes de correção e performance:

Testes de performance de $optimize\_eval$:

$$sums = (\!|[\underline{(N\ 1)}, Bin\ Sum\ (N\ 3)]|\!)$$
$$p = Bin\ Product\ (sums\ 1000)\ (N\ 0)$$

```
*Main> eval_exp 1 p
0.0
(0.04 secs, 4,098,576 bytes)
*Main> eval_exp_int 1 p
0.0
(0.05 secs, 4,098,768 bytes)
*Main> optmize_eval 1 p
0.0
(0.02 secs, 368,704 bytes)
```

Problema 2

Definir

$$\begin{array}{ll} loop\ (c, t, b) & = (t * c \div b, 4 + t, 1 + b) \\ inic & = (1, 2, 2) \\ prj\ \ (c, \_, \_) & = c \end{array}$$

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.

Fórmula que dá o n-ésimo número de Catalan:

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

Funções para recursividade mútua:

$$\begin{array}{rcl} c\, n & \stackrel{\mathrm{def}}{=} & \dfrac{(2n)!}{(n+1)!(n!)} \\ c\, 0 & = & 1 \\ c\, (n+1) & = & \dfrac{4n+2}{n+2}(c\, n) \end{array}$$

$$\begin{array}{rcl} t\, n & \stackrel{\mathrm{def}}{=} & 4n+2 \\ t\, 0 & = & 2 \\ t\, (n+1) & = & 4 + t\, n \end{array}$$

$$\begin{array}{rcl} b\, n & \stackrel{\mathrm{def}}{=} & n+2 \\ b\, 0 & = & 2 \\ b\, (n+1) & = & 1 + b\, n \end{array}$$

Redefinindo c,

$$\begin{array}{rcl} c\, 0 & = & 1 \\ c\, (n+1) & = & \dfrac{t\, n}{b\, n}(c\, n) \\ & = & \dfrac{(t\, n)(c\, n)}{b\, n} \end{array}$$

Das definições das funções c, t e b é usada a *regra de algibeira* descrita na página 3.1 deste enunciado para derivar uma implementação de C_n

Desenvolvimento das expressões algébricas acima:

$$
\begin{aligned}
c\,0 &= \frac{(2*0)!}{(0+1)!(0!)} = \frac{0!}{1!*1} = \frac{1}{1} = 1 \\
c\,(n+1) &= \frac{(2(n+1))!}{((n+1)+1)!((n+1)!)} \\
&= \frac{(2n+2)!}{(n+2)!(n+1)!} \\
&= \frac{(2n+2)(2n+1)(2n)!}{(n+2)(n+1)!(n+1)n!} \\
&= \frac{(2n+2)(2n+1)}{(n+2)(n+1)} \cdot \frac{(2n)!}{(n+1)!n!} \\
&= \frac{4n+2}{n+2}(c\,n) \\
t\,0 &= 4*0+2 = 0+2 = 2 \\
t\,(n+1) &= 4(n+1)+2 \\
&= 4n+4+2 \\
&= 4+(4n+2) \\
&= 4+tn \\
b\,0 &= 0+2 = 2 \\
b\,(n+1) &= (n+1)+2 \\
&= 1+(n+2) \\
&= 1+b\,n
\end{aligned}
$$

Testes de correção e performance:

$$
\begin{aligned}
&oracleCmp = (\mathsf{map}\ \ cat\ [0\,..\,25]) \equiv oracle \\
&catdefCmp = (\mathsf{map}\ \ cat\ [0\,..\,99]) \equiv (\mathsf{map}\ \ catdef\ [0\,..\,99])
\end{aligned}
$$

```
*Main> oracleCmp
True
*Main> catdefCmp
True
*Main> catdef 100000
1780545081823061907837573390658902019302...74049460495513844450580552321237059507840
(68.63 secs, 71,864,876,408 bytes)
*Main> cat 100000
1780545081823061907837573390658902019302...74049460495513844450580552321237059507840
(4.48 secs, 3,077,657,400 bytes)
```

Problema 3

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

$$
\begin{array}{l}
deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint \\
deCasteljau = [\![\, conquer, divide \,]\!]_{Bezier}\ \textbf{where} \\
\quad divide \quad = (id + (id + \langle init, tail \rangle)) \cdot \mathsf{out}_{Bezier} \\
\quad conquer = [\underline{nil}, [\underline{\cdot}, f]] \\
\quad f\ (a, b) \ \ = \lambda pt \rightarrow (calcLine\ (a\ pt)\ (b\ pt))\ pt
\end{array}
$$

$$
\begin{array}{l}
\mathsf{out}_{Bezier}\ [\,] \quad = i_1\ () \\
\mathsf{out}_{Bezier}\ [a] = i_2\ \$\ i_1\ a \\
\mathsf{out}_{Bezier}\ l \quad = i_2\ \$\ i_2\ l \\
\quad \text{--} \\
[\![\, f, g \,]\!]_{Bezier} = f \cdot \mathsf{F}_{Bezier}\ [\![\, f, g \,]\!]_{Bezier} \cdot g \\
\quad \text{--} \\
\mathsf{F}_{Bezier}\ f = id + (id + f \times f)
\end{array}
$$

Prova da definição de calcLine

$$
\begin{tikzcd}
NPoint \arrow[rr, "\mathsf{out}_{List}"] \arrow[d, "(\!| h |\!)"'] & & 1 + \mathbb{Q} + NPoint \arrow[d, "1 + id \times (\!| h |\!)"] \\
(Overtime\ NPoint)^{NPoint} & & 1 + \mathbb{Q} + (Overtime\ NPoint)^{NPoint} \arrow[ll, "h"]
\end{tikzcd}
$$

$$
\begin{aligned}
& \begin{cases} calcLine\ [\,] = \underline{nil} \\ calcLine\ (p : x) = \overline{g}\ p\ (calcLine\ x) \end{cases} \\
\equiv \quad & \{ \text{ Def-comp } \} \\
& \begin{cases} calcLine \cdot nil\ a = \underline{\underline{nil}}\ a \\ calcLine \cdot cons\ (p, x) = g \cdot (id \times calcLine)\ (p, x) \end{cases} \\
\equiv \quad & \{ \text{ Igualdade extensional } \} \\
& \begin{cases} calcLine \cdot nil = \underline{\underline{nil}} \\ calcLine \cdot cons = g \cdot (id \times calcLine) \end{cases} \\
\equiv \quad & \{ \text{ Eq-+; Natural-id } \} \\
& [calcLine \cdot nil, calcLine \cdot cons] = [\underline{\underline{nil}}, g \cdot (id \times calcLine)] \\
\equiv \quad & \{ \text{ Fusão-+; Absorção-+ } \} \\
& calcLine \cdot [nil, cons] = [\underline{\underline{nil}}, g] \cdot (id + id \times calcLine)) \\
\equiv \quad & \{ \text{ Def-inList; Def-F } \} \\
& calcLine \cdot \mathsf{in}_{List} = [\underline{\underline{nil}}, g] \cdot \mathsf{F}\ calcLine \\
\equiv \quad & \{ \text{ Universal-cata } \} \\
& calcLine = (\!| [\underline{\underline{nil}}, g] |\!) \\
\square \quad &
\end{aligned}
$$

Conclui-se assim que $h = [cons\ (cons\ nil), g]$ onde g é definido por:

$$
\begin{array}{l}
g\ (d, f)\ l = \mathbf{case}\ l\ \mathbf{of} \\
\quad [\,] \rightarrow nil \\
\quad (x : xs) \rightarrow \lambda z \rightarrow concat\ \$\ (sequenceA\ [singl \cdot linear1d\ d\ x, f\ xs])\ z
\end{array}
$$

Prova da definição de deCasteljau

$$\left\{\begin{array}{l} deCasteljau\ [\,] = nil \\ deCasteljau\ [p] = \underline{p} \\ deCasteljau\ l = \lambda pt \rightarrow (calcLine\ (p\ pt)\ (q\ pt))\ pt \end{array}\right.$$

$\equiv$ { p = deCasteljau (init l); q = deCasteljau (tail l) }

$$\left\{\begin{array}{l} deCasteljau\ [\,] = nil \\ deCasteljau\ [p] = \underline{p} \\ deCasteljau\ l = \lambda pt \rightarrow (calcLine\ (deCasteljau\ (init\ l)\ pt)\ (deCasteljau\ (tail\ l)\ pt))\ pt \end{array}\right.$$

$\equiv$ { Igualdade extensional; Def-comp }

$$\left\{\begin{array}{l} deCasteljau \cdot nil = \underline{nil} \\ deCasteljau \cdot singl = const \\ deCasteljau = \widehat{calcLine} \cdot deCasteljau^2 \cdot \langle init, tail \rangle \end{array}\right.$$

$\equiv$ { $2 \times$ Eq-+; $2 \times$ Fusão-+; $2 \times$ Absorção-+ }

$$deCasteljau \cdot [nil, [singl, id]] = [\underline{nil}, [const, \widehat{calcLine}]] \cdot (id + (id + deCasteljau^2)) \cdot (id + (id + \langle init, tail \rangle))$$

$\equiv$ { Shunt-left }

$$deCasteljau = [\underline{nil}, [const, \widehat{calcLine}]] \cdot (id + (id + deCasteljau^2)) \cdot (id + (id + \langle init, tail \rangle)) \cdot \mathsf{out}_{Bezier}$$

$\square$

Esta prova não é satisfatória para definir $deCasteljau$ como um hylomorfismo devido à dificuldade de proceder nesta com a função anónima. Porém, seguindo esta prova e outros exemplos da aula 9 da disciplina é possivél concluir que:

- $divide = (id + (id + \langle init, tail \rangle)) \cdot \mathsf{out}_{Bezier}$
- $conquer = [\underline{nil}, [\underline{\cdot}, f]]$
- $\mathsf{F}_{Bezier}\ f = id + (id + f \times f)$

Onde a função f em $conquer$ é definida por:

- $f\ (a, b) = \lambda pt \rightarrow (calcLine\ (a\ pt)\ (b\ pt))\ pt$

Testes de correção

Definições das funções do Problema 3 dadas como especificações:

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

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

Funções de verificação das funções definidas como resposta ao Problema 3 através das especificações destas:

$$
\begin{array}{l}
verifyCalcLine\ pt1\ pt2\ x = (calcLine\ pt1\ pt2\ x) \equiv (calcLineSpec\ pt1\ pt2\ x) \\
verifyDeCasteljau\ pts\ x = (deCasteljau\ pts\ x) \equiv (deCasteljauSpec\ pts\ x)
\end{array}
$$

Verificação no ghci:

```
*Main> verifyCalcLine [0,0] [0,1] 0.5
True
*Main> verifyDeCasteljau [[0,0],[0,1],[1,0]] 0.5
True
*Main> map fromRational $ deCasteljau [[0,0],[0,1],[1,0]] 0.5
[0.25,0.5]
```

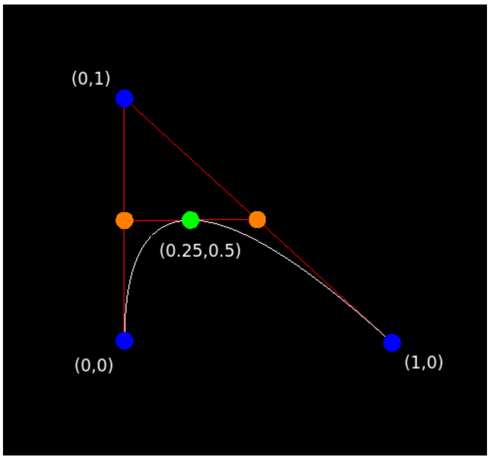

Figura 2: Exemplo de curva de Bézier criada com as funções dadas como especificação.

Problema 4

Solução para listas não vazias:

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

$$\begin{array}{l} avg\_aux = (\![[init, loop]]\!)_{NList} \ \mathbf{where} \\ \quad loop\ (a, (b, c)) = ((a + b * c) \mathbin{/} (c + 1), c + 1) \\ \quad init\ \ a \qquad\qquad = (a, 1) \end{array}$$

Solução para árvores de tipo LTree:

$$\begin{array}{l} avgLTree = \pi_1 \cdot (\![gene]\!) \ \mathbf{where} \\ \quad gene = [init, loop] \\ \quad loop\ ((a_1, b_1), (a_2, b_2)) = ((a_1 * b_1 + a_2 * b_2) \mathbin{/} (b_1 + b_2), b_1 + b_2) \\ \quad init\ \ a \qquad\qquad\qquad\quad = (a, 1) \end{array}$$

Definições de funções para catamorfismos sobre listas não vazias:

$$\begin{array}{l} \mathsf{in}_{NList} :: a + (a, [a]) \rightarrow [a] \\ \mathsf{in}_{NList} = [singl, cons] \\ \quad \text{--} \\ \mathsf{out}_{NList} :: [a] \rightarrow a + (a, [a]) \\ \mathsf{out}_{NList}\ [a] \qquad = i_1\ a \\ \mathsf{out}_{NList}\ (a : as) = i_2\ (a, as) \\ \quad \text{--} \\ (\![\cdot]\!)_{NList} :: (a + (a, b) \rightarrow b) \rightarrow [a] \rightarrow b \\ (\![g]\!)_{NList} = g \cdot \mathsf{F}_{NList}\ (\![g]\!)_{NList} \cdot \mathsf{out}_{NList} \\ \quad \text{--} \\ \mathsf{F}_{NList} :: (a \rightarrow b) \rightarrow x + (y, a) \rightarrow x + (y, b) \\ \mathsf{F}_{NList}\ f = id + id \times f \end{array}$$

Prova da definição de out_{NList}

$$\begin{tikzcd} A + A \times A^{+} \arrow[rr, bend left, "\mathsf{out}_{NList}"'] \arrow[rr, phantom, "\cong"] & & A^{+} \arrow[ll, bend left, "\mathsf{in}_{NList}"'] \end{tikzcd}$$

$$\begin{array}{ll}
& \mathsf{out}_{NList} \cdot \mathsf{in}_{NList} = id \\
\equiv & \quad \{ \text{ Def-in}_{NList} \text{ } \} \\
& \mathsf{out}_{NList} \cdot [singl, cons] = id \\
\equiv & \quad \{ \text{ Fusão-+ } \} \\
& [\mathsf{out}_{NList} \cdot singl, \mathsf{out}_{NList} \cdot cons] = id \\
\equiv & \quad \{ \text{ Universal-+; Natural-id } \} \\
& \left\{ \begin{array}{l} \mathsf{out}_{NList} \cdot singl = i_1 \\ \mathsf{out}_{NList} \cdot cons = i_2 \end{array} \right. \\
\equiv & \quad \{ \text{ Igualdade extensional; Def-comp } \} \\
& \left\{ \begin{array}{l} \mathsf{out}_{NList}\ (singl\ a) = i_1\ a \\ \mathsf{out}_{NList}\ (cons\ (a, as)) = i_2\ (a, as) \end{array} \right. \\
\equiv & \quad \{ \text{ Def-singl; Def-cons } \} \\
& \left\{ \begin{array}{l} \mathsf{out}_{NList}\ [a] = i_1\ a \\ \mathsf{out}_{NList}\ (a : as) = i_2\ (a, as) \end{array} \right. \\
\Box &
\end{array}$$

Prova da definição do gene de avg_aux

$$
\begin{tikzcd}
A^+ \arrow[d, "avg\_aux"'] & & A + A \times A^+ \arrow[ll, "\mathsf{in}_{NList}"'] \arrow[d, "id + id \times avg\_aux"] \\
A \times \mathbb{N}^+ & & A + A \times (A \times \mathbb{N}^+) \arrow[ll, "{[b, q]}"]
\end{tikzcd}
$$

$$
\begin{array}{ll}
& avg\_aux = (\!|\, [b, q] \,|\!)_{NList} \\
\equiv & \quad \{ \ avg\_aux = \langle avg, length \rangle \ \} \\
& \langle avg, length \rangle = (\!|\, [b, q] \,|\!)_{NList} \\
\equiv & \quad \{ \ \text{Inferência dos tipos de b e q} \ \} \\
& \langle avg, length \rangle = (\!|\, [\langle b_1, b_2 \rangle, \langle q_1, q_2 \rangle] \,|\!)_{NList} \\
\equiv & \quad \{ \ \text{Lei da troca} \ \} \\
& \langle avg, length \rangle = (\!|\, \langle [b_1, q_1], [b_2, q_2] \rangle \,|\!)_{NList} \\
\equiv & \quad \{ \ \text{Lei da recursividade mútua (Fokkinga)} \ \} \\
& \left\{ \begin{array}{l} avg \cdot \mathsf{in}_{NList} = [b_1, q_1] \cdot \mathsf{F}\ \langle avg, length \rangle \\ length \cdot \mathsf{in}_{NList} = [b_2, q_2] \cdot \mathsf{F}\ \langle avg, length \rangle \end{array} \right. \\
\equiv & \quad \{ \ \text{Def-inNList; Def-}\mathsf{F} \ \} \\
& \left\{ \begin{array}{l} avg \cdot [singl, cons] = [b_1, q_1] \cdot (id + id \times \langle avg, length \rangle) \\ length \cdot [singl, cons] = [b_2, q_2] \cdot (id + id \times \langle avg, length \rangle) \end{array} \right. \\
\equiv & \quad \{ \ 2 \times \text{Fusão-+}; \ 2 \times \text{Absorção-+}; \ 2 \times \text{Natural-id} \ \} \\
& \left\{ \begin{array}{l} [avg \cdot singl, avg \cdot cons] = [b_1, q_1 \cdot (id \times \langle avg, length \rangle)] \\ [length \cdot singl, length \cdot cons] = [b_2, q_2 \cdot (id \times \langle avg, length \rangle)] \end{array} \right. \\
\equiv & \quad \{ \ 2 \times \text{Eq-+}; \ \text{f = g} \equiv \text{g = f} \ \} \\
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} b_1 = avg \cdot singl \\ q_1 \cdot (id \times \langle avg, length \rangle) = avg \cdot cons \end{array} \right. \\ \left\{ \begin{array}{l} b_2 = length \cdot singl \\ q_2 \cdot (id \times \langle avg, length \rangle) = length \cdot cons \end{array} \right. \end{array} \right. \\
\equiv & \quad \{ \ \text{Igualdade extensional; Def-comp} \ \} \\
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} b_1 \ a = avg \ (singl \ a) \\ q_1 \ ((id \times \langle avg, length \rangle) \ (a, as)) = avg \ (cons \ (a, as)) \end{array} \right. \\ \left\{ \begin{array}{l} b_2 \ a = length \ (singl \ a) \\ q_2 \ ((id \times \langle avg, length \rangle) \ (a, as)) = length \ (cons \ (a, as)) \end{array} \right. \end{array} \right. \\
\equiv & \quad \{ \ \text{Def-cons; Def-}\times\text{; Def-split; Def-singl; Natural-id} \ \} \\
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} b_1 \ a = avg \ [a] \\ q_1 \ (a, (avg \ as, length \ as)) = avg \ (a : as) \end{array} \right. \\ \left\{ \begin{array}{l} b_2 \ a = length \ [a] \\ q_2 \ (a, (avg \ as, length \ as)) = length \ (a : as) \end{array} \right. \end{array} \right. \\
\equiv & \quad \{ \ \text{Def-avg; Def-length} \ \} \\
& \left\{ \begin{array}{l} \left\{ \begin{array}{l} b_1 \ a = a \\ q_1 \ (a, (avg \ as, length \ as)) = (a + avg \ as * length \ as) \ / \ (legth \ as + 1) \end{array} \right. \\ \left\{ \begin{array}{l} b_2 \ a = 1 \\ q_2 \ (a, (avg \ as, length \ as)) = length \ as + 1 \end{array} \right. \end{array} \right.
\end{array}
$$

$\square$

Prova da definição do gene de avgLTree

$$\begin{tikzcd}
\mathsf{LTree}\ A \arrow[d, "(\![b,q]\!)"'] & A + (\mathsf{LTree}\ A)^2 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + (\![b,q]\!)^2"] \\
A \times \mathbb{N}^+ & A + (A \times \mathbb{N}^+)^2 \arrow[l, "{[b,q]}"]
\end{tikzcd}$$

$$
\begin{array}{ll}
& avgLTree = (\![[b, q]]\!) \\
\equiv & \quad \{ \ avgLTree = \langle avg, length \rangle \ \} \\
& \langle avg, length \rangle = (\![[b, q]]\!) \\
\equiv & \quad \{ \ \text{Inferência dos tipos de b e q} \ \} \\
& \langle avg, length \rangle = (\![[\langle b_1, b_2 \rangle, \langle q_1, q_2 \rangle]]\!) \\
\equiv & \quad \{ \ \text{Lei da troca} \ \} \\
& \langle avg, length \rangle = (\![\langle [b_1, q_1], [b_2, q_2] \rangle]\!) \\
\equiv & \quad \{ \ \text{Lei da recursividade mútua (Fokkinga)} \ \} \\
& \begin{cases} avg \cdot \mathsf{in} = [b_1, q_1] \cdot \mathsf{F}\ \langle avg, length \rangle \\ length \cdot \mathsf{in} = [b_2, q_2] \cdot \mathsf{F}\ \langle avg, length \rangle \end{cases} \\
\equiv & \quad \{ \ \text{Def-in; Def-F} \ \} \\
& \begin{cases} avg \cdot [Leaf, Fork] = [b_1, q_1] \cdot (id + \langle avg, length \rangle^2) \\ length \cdot [Leaf, Fork] = [b_2, q_2] \cdot (id + \langle avg, length \rangle^2) \end{cases} \\
\equiv & \quad \{ \ 2 \times \text{Fusão-+}; \ 2 \times \text{Absorção-+}; \ 2 \times \text{Natural-id} \ \} \\
& \begin{cases} [avg \cdot Leaf, avg \cdot Fork] = [b1, q_1 \cdot \langle avg, length \rangle^2] \\ [length \cdot Leaf, length \cdot Fork] = [b2, q_2 \cdot \langle avg, length \rangle^2] \end{cases} \\
\equiv & \quad \{ \ 2 \times \text{Eq-+}; \ \text{f = g} \equiv \text{g = f} \ \} \\
& \begin{cases} \begin{cases} b_1 = avg \cdot Leaf \\ q_1 \cdot \langle avg, length \rangle^2 = avg \cdot Fork \end{cases} \\ \begin{cases} b_2 = length \cdot Leaf \\ q_2 \cdot \langle avg, length \rangle^2 = length \cdot Fork \end{cases} \end{cases} \\
\equiv & \quad \{ \ \text{Igualdade extensional; Def-comp} \ \} \\
& \begin{cases} \begin{cases} b_1\ a = avg\ (Leaf\ \ a) \\ q_1\ (\langle avg, length \rangle^2\ ((a_1, b_1), (a_2, b_2))) = avg\ (Fork\ ((a_1, b_1), (a_2, b_2))) \end{cases} \\ \begin{cases} b_2\ a = length\ (Leaf\ \ a) \\ q_2\ (\langle avg, length \rangle^2\ ((a_1, b_1), (a_2, b_2))) = length\ (Fork\ ((a_1, b_1), (a_2, b_2))) \end{cases} \end{cases} \\
\equiv & \quad \{ \ \text{Def-Leaf; Def-Fork; Def-}\times\text{; Def-split; Natural-id} \ \} \\
& \begin{cases} \begin{cases} b_1\ a = avg\ (Leaf\ \ a) \\ q_1\ ((avg\ (a_1, b_1), length\ (a_1, b_1)), (avg\ (a_2, b_2), length\ (a_2, b_2))) = avg\ (Fork\ ((a_1, b_1), (a_2, b_2))) \end{cases} \\ \begin{cases} b_2\ a = length\ (Leaf\ \ a) \\ q_2\ ((avg\ (a_1, b_1), length\ (a_1, b_1)), (avg\ (a_2, b_2), length\ (a_2, b_2))) = length\ (Fork\ ((a_1, b_1), (a_2, b_2))) \end{cases} \end{cases} \\
\equiv & \quad \{ \ \text{Def-avg; Def-length} \ \} \\
& \begin{cases} \begin{cases} b_1\ a = a \\ q_1\ ((a_1, b_1), (a_2, b_2)) = (a_1 * b_1 + a_2 * b_2)\ /\ (b_1 + b_2) \end{cases} \\ \begin{cases} b_2\ a = 1 \\ q_2\ ((a_1, b_1), (a_2, b_2)) = b_1 + b_2 \end{cases} \end{cases}
\end{array}
$$

$\square$

Testes de correção

Por 2 podemos definir uma função que cálcula a média aritmética de uma lista como:

$$avgListDef :: (Fractional\ a, Num\ a) \Rightarrow [a] \to a$$
$$avgListDef = \widehat{(/)} \cdot \langle sum, fromIntegral \cdot length \rangle$$

Uma função que cálcula a média aritmética de uma LTree pode ser definida como a função que cálcula a média de uma lista após converter a LTree para tal lista através da função $tips$ definida em $Cp.hs$:

$$avgLTreeDef :: (Fractional\ a, Num\ a) \Rightarrow \mathsf{LTree}\ a \to a$$
$$avgLTreeDef = avgListDef \cdot tips$$

Funções de verificação das funções definidas como resposta ao Problema 4 através das definições feitas acimas:

$$verifyAvgList = (\leqslant 0.000001) \cdot \widehat{(-)} \cdot \langle avgListDef, avg \rangle$$
$$verifyAvgLTree = (\leqslant 0.000001) \cdot \widehat{(-)} \cdot \langle avgLTreeDef, avgLTree \rangle \cdot genLTree\ \mathbf{where}$$
$$\quad genLTree = [\!(\, lsplit \,)\!]$$

Verificação no ghci:

```
*Main> verifyAvgList [1.0, 1.3 .. 100]
True
*Main> verifyAvgLTree [1.0, 1.3 .. 100]
True
```

Problema 5

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

```
module cp2021t

open Cp
//import Data.List
//import Data.Monoid

// (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,(b1,b2)) -> i2 (a,(b1,b2))

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

// recBTree g = id -|- (id >< (g >< g))

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 )

// equivalent to:
//       where fmap f = anaBTree ( baseBTree f id . outBTree )

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

// (4.1) Inversion (mirror) ----------------------------------------------------

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

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

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

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

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

```
        in either nil join x

let inordt x = cataBTree insord  x                 // in-order traversal

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 menor x z = z < x

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) = part p


let qsep x =
    match x with
    | [] -> Left ()
    | (h::t) -> Right (h,(part (menor h) t))

let qSort  x = hyloBTree insord qsep x // the same as (cataBTree inord) . (anaBTree q


(* pointwise versions:
qSort [] = []
qSort (h:t) = let (t1,t2) = part (<h) t
              in  qSort t1 ++ [h] ++ qSort t2

or, using list comprehensions:

qSort [] = []
qSort (h:t) = qSort [ a | a <- t , a < h ] ++ [h] ++
              qSort [ a | a <- t , a >= h ]

*)

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

let cons x z = x::z

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

let rec union l ls =
    match ls with
    | [] -> l
    | (h::t) -> if elem h l then union l t else (union l t) @ [h]
```

```
let tunion (a,(l,r)) = union (List.map (cons a) l) (List.map (cons 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 present x = insord x // same as in qSort

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

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 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 baldepth x =
    let g = either (konst(true,1)) (h << (id><f))
    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
tnat :: Monoid c => (a -> c) -> Either () (a,(c, c)) -> c
tnat f = either (const mempty) (theta . (f >< theta))
         where theta = uncurry mappend

-- monoid reduction

monBTree f = cataBTree (tnat f)

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

preordt' = monBTree singl

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

countBTree' = monBTree (const (Sum 1))

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

data Deriv a = Dr Bool a (BTree a)

type Zipper a = [ Deriv a ]

plug :: Zipper a -> BTree a -> BTree a
plug [] t = t
plug ((Dr False a l):z) t = Node (a,(plug z t,l))
plug ((Dr True  a r):z) t = Node (a,(r,plug z t))

---------------------------- end of library ----------------------------------
*)
```

Outras soluções

-- Definição point free de g_eval_exp

$$
\begin{aligned}
&g\_eval\_exp\_pf\ v = [\underline{v}, [id, [bin, un]]]\ \textbf{where}\\
&\quad bin \qquad\qquad = ap \cdot (binop \times id)\\
&\quad un \qquad\qquad = ap \cdot (unop\ \times id)\\
&\quad binop\ Sum \quad\ = addP\\
&\quad binop\ Product = mulP\\
&\quad unop\ \ Negate\ \ = negate\\
&\quad unop\ \ E \qquad\ \ = expd
\end{aligned}
$$

--

-- Definição point wise de g_eval_exp com condicionais

$$
\begin{aligned}
&g\_eval\_exp\_cpw\ v = [g_1, [g_2, [g_3, g_4]]]\ \textbf{where}\\
&\quad g_1\ () = v\\
&\quad g_2\ a = a\\
&\quad g_3\ (binop, (a, b)) \mid binop \equiv Sum = a + b\\
&\qquad\qquad\qquad\qquad\ \mid otherwise \quad\ = a * b\\
&\quad g_4\ (unop, a) \mid unop \equiv Negate = negate\ a\\
&\qquad\qquad\qquad\ \mid otherwise \qquad\ \ = expd\ a
\end{aligned}
$$

--

-- Definição point free de g_eval_exp com condicionais

$$
\begin{aligned}
&g\_eval\_exp\_cpf\ v = [g_1, [g_2, [g_3, g_4]]]\ \textbf{where}\\
&\quad g_1 = \underline{v}\\
&\quad g_2 = id\\
&\quad g_3 = ((Sum \equiv) \cdot \pi_1) \to (addP \cdot \pi_2), (mulP \cdot \pi_2)\\
&\quad g_4 = ((Negate \equiv) \cdot \pi_1) \to (negate \cdot \pi_2), (expd \cdot \pi_2)
\end{aligned}
$$

-- Definição point free de sd_gen

$$
\begin{aligned}
&sd\_gen\_pf :: Floating\ a \Rightarrow\\
&\quad () + (a + ((BinOp, ((ExpAr\ a, ExpAr\ a), (ExpAr\ a, ExpAr\ a))) + (UnOp, (ExpAr\ a, ExpAr\ a))))\\
&\quad\ \to (ExpAr\ a, ExpAr\ a)\\
&sd\_gen\_pf = [\underline{(X, N\ 1)}, [\langle N, \underline{(N\ 0)}\rangle, [bin, un]]]\ \textbf{where}\\
&\quad bin = ap \cdot (binop \times id)\\
&\quad un\ = ap \cdot (unop\ \times id)\\
&\quad binop\ Sum \quad\ = \langle (\widehat{Bin\ Sum}) \cdot (\pi_1 \times \pi_1), (\widehat{Bin\ Sum}) \cdot (\pi_2 \times \pi_2)\rangle\\
&\quad binop\ Product = \langle (\widehat{Bin\ Product}) \cdot (\pi_1 \times \pi_1), (\widehat{Bin\ Sum}) \cdot \langle (\widehat{Bin\ Product}) \cdot (\pi_1 \times \pi_2), (\widehat{Bin\ Product}) \cdot (\pi_2 \times \pi_1)\rangle\\
&\quad unop\ \ Negate\ \ = (Un\ Negate \times Un\ Negate)\\
&\quad unop\ \ E \qquad\ \ = \langle Un\ E \cdot \pi_1, (\widehat{Bin\ Product}) \cdot (Un\ E \times id)\rangle
\end{aligned}
$$

-- Definição point free de ad_gen

$$
\begin{aligned}
&ad\_gen\_pf\ v = [\underline{(v, 1)}, [\langle id, \underline{0}\rangle, [bin, un]]]\ \textbf{where}\\
&\quad bin = ap \cdot (binop \times id)\\
&\quad un\ = ap \cdot (unop\ \times id)\\
&\quad binop\ Sum \quad\ = \langle addP \cdot (\pi_1 \times \pi_1), addP \cdot (\pi_2 \times \pi_2)\rangle\\
&\quad binop\ Product = \langle mulP \cdot (\pi_1 \times \pi_1), addP \cdot \langle mulP \cdot (\pi_1 \times \pi_2), mulP \cdot (\pi_2 \times \pi_1)\rangle\rangle\\
&\quad unop\ \ Negate\ \ = (negate \times negate)\\
&\quad unop\ \ E \qquad\ \ = \langle expd \cdot \pi_1, mulP \cdot (expd \times id)\rangle
\end{aligned}
$$

-- Definição point free de avg_aux

$$
avg\_aux\_pf = (\![\, [\langle id, oneP\rangle, \langle \widehat{(/)} \cdot \langle addP \cdot (id \times mulP), \mathsf{succ} \cdot \pi_2 \cdot \pi_2\rangle, \mathsf{succ} \cdot \pi_2 \cdot \pi_2\rangle] \,]\!)_{NList}
$$

--

-- Definição point free de avgLTree

$$
\begin{aligned}
&avgLTree\_pf = \pi_1 \cdot (\![\, gene \,]\!)\ \textbf{where}\\
&\quad gene = [\langle id, oneP\rangle, \langle \widehat{(/)} \cdot \langle addP \cdot (mulP \times mulP), addP \cdot (\pi_2 \times \pi_2)\rangle, addP \cdot (\pi_2 \times \pi_2)\rangle]
\end{aligned}
$$

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.