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

Departamento de Informática
Universidade do Minho

Maio de 2022

Grupo nr.	17
a91683	Alef Pinto Keuffer
a93546	Fernando Maria Bicalho
a88062	Pedro Paulo Costa Pereira
a91693	Tiago André Oliveira Leite

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 depêndencias externas serão instaladas automaticamente.

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

Problema 1

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

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

- *Symbolic differentiation*
- *Automatic differentiation*

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

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

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

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

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

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

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 *inExpAr* e *baseExpAr* para este tipo é a seguinte:

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

Defina as funções *outExpAr* e *recExpAr*, e teste as propriedades que se seguem.

Propriedade [**QuickCheck**] **1** *inExpAr e 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

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

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

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

$prop\_const\_rule :: (Real\ a, Floating\ a) \Rightarrow a \to Bool$
$prop\_const\_rule\ a = sd\ (N\ a) \equiv N\ 0$
$prop\_var\_rule :: Bool$
$prop\_var\_rule = sd\ X \equiv N\ 1$
$prop\_sum\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \to ExpAr\ a \to Bool$
$prop\_sum\_rule\ exp1\ exp2 = sd\ (Bin\ Sum\ exp1\ exp2) \equiv sum\_rule$ **where**
$\quad sum\_rule = Bin\ Sum\ (sd\ exp1)\ (sd\ exp2)$
$prop\_product\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \to ExpAr\ a \to Bool$
$prop\_product\_rule\ exp1\ exp2 = sd\ (Bin\ Product\ exp1\ exp2) \equiv prod\_rule$ **where**
$\quad prod\_rule = Bin\ Sum\ (Bin\ Product\ exp1\ (sd\ exp2))\ (Bin\ Product\ (sd\ exp1)\ exp2)$
$prop\_e\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \to Bool$
$prop\_e\_rule\ exp = sd\ (Un\ E\ exp) \equiv Bin\ Product\ (Un\ E\ exp)\ (sd\ exp)$
$prop\_negate\_rule :: (Real\ a, Floating\ a) \Rightarrow ExpAr\ a \to Bool$
$prop\_negate\_rule\ exp = sd\ (Un\ Negate\ exp) \equiv Un\ Negate\ (sd\ exp)$

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

   Defina o gene do catamorfismo que ocorre na função

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

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

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

   $prop\_congruent :: (Floating\ a, Real\ a) \Rightarrow a \to ExpAr\ a \to Bool$
   $prop\_congruent\ a\ exp = ad\ a\ exp \stackrel{?}{=} eval\_exp\ a\ (sd\ exp)$

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

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

[4] Lei (3.94) em [2], página 98.

$$f\ 0 = 1$$
$$f\ (n+1) = \mathit{fib}\ n + f\ n$$

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:*

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

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

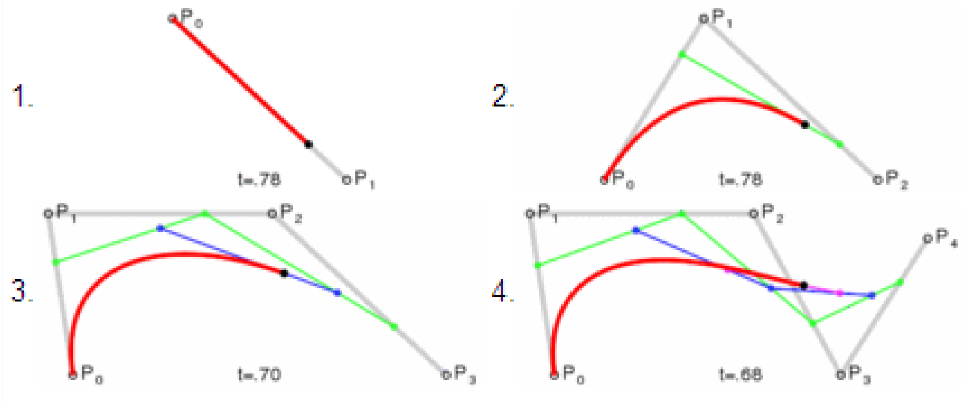

Figura 1: Exemplos de curvas de Bézier retirados da Wikipedia.

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:

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

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.

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

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 $Float$).

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

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.*

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

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:*

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

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 [2].
[9]Cf. [2], página 102.

C Código fornecido

Problema 1

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

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]:

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

Problema 3

Algoritmo:

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

Função auxiliar:

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

2D:

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

Modelo:

$$
\begin{aligned}
&\textbf{data}\ \mathit{World} = \mathit{World}\ \{\mathit{points} :: [\mathit{NPoint}]\\
&\quad , \mathit{time} :: \mathit{Float}\\
&\quad \}\\
&\mathit{initW} :: \mathit{World}\\
&\mathit{initW} = \mathit{World}\ [\,]\ 0\\
&\mathit{tick} :: \mathit{Float} \rightarrow \mathit{World} \rightarrow \mathit{World}
\end{aligned}
$$

[10] Fonte: Wikipedia.

$tick\ dt\ world = world\ \{ time = (time\ world) + dt \}$

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

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

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

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

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

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

Gloss:

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

Animação:

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

Propriedades e *main*:

$runBezier :: \mathsf{IO}\ ()$
$runBezier = play\ (InWindow$ `"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]

$main = runBezier$

$run =$ **do** $\{ system$ `"ghc cp2021t"`; $system$ `"./cp2021t"` $\}$

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

QuickCheck

Código para geração de testes:

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{UnOp}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Negate}, E] \\
\textbf{instance}\ \mathit{Arbitrary}\ \mathit{BinOp}\ \textbf{where} \\
\quad \mathit{arbitrary} = \mathit{elements}\ [\mathit{Sum}, \mathit{Product}] \\
\textbf{instance}\ (\mathit{Arbitrary}\ a) \Rightarrow \mathit{Arbitrary}\ (\mathit{ExpAr}\ a)\ \textbf{where} \\
\quad \mathit{arbitrary} = \textbf{do} \\
\qquad \mathit{binop} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{unop} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{exp1} \leftarrow \mathit{arbitrary} \\
\qquad \mathit{exp2} \leftarrow \mathit{arbitrary} \\
\qquad a \quad\ \ \leftarrow \mathit{arbitrary} \\
\qquad \mathit{frequency} \cdot \mathsf{map}\ \ (\mathit{id} \times \mathit{pure})\ \$\ [(20, X), (15, N\ a), (35, \mathit{Bin}\ \mathit{binop}\ \mathit{exp1}\ \mathit{exp2}), (30, \mathit{Un}\ \mathit{unop}\ \mathit{exp1})] \\
\textbf{infixr}\ 5\ \stackrel{?}{=} \\
(\stackrel{?}{=}) :: \mathit{Real}\ a \Rightarrow a \rightarrow a \rightarrow \mathit{Bool} \\
(\stackrel{?}{=})\ x\ y = (\mathit{to}_{\mathbb{Q}}\ x) \equiv (\mathit{to}_{\mathbb{Q}}\ y)
\end{array}
$$

Outras funções auxiliares

Lógicas:

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

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:

$$
\begin{array}{l}
\mathit{cataExpAr}\ g = g \cdot \mathit{recExpAr}\ (\mathit{cataExpAr}\ g) \cdot \mathit{outExpAr} \\
\mathit{anaExpAr}\ g = \mathit{inExpAr} \cdot \mathit{recExpAr}\ (\mathit{anaExpAr}\ g) \cdot g \\
\mathit{hyloExpAr}\ h\ g = \mathit{cataExpAr}\ h \cdot \mathit{anaExpAr}\ g \\
\mathit{eval\_exp} :: \mathit{Floating}\ a \Rightarrow a \rightarrow (\mathit{ExpAr}\ a) \rightarrow a
\end{array}
$$

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

$$
\begin{array}{ll}
 & outExpAr \cdot inExpAr = id \\
\equiv & \quad \{ \text{ Definição de inExpAr; Fusão-+; Cancelamento-+ } \} \\
 & \left\{ \begin{array}{l} outExpAr \cdot \underline{X} = id \cdot i_1 \\ \left\{ \begin{array}{l} outExpAr \cdot N = id \cdot i_2 \cdot i_1 \\ \left\{ \begin{array}{l} outExpAr \cdot bin = id \cdot i_2 \cdot i_2 \cdot i_1 \\ outExpAr \cdot \widehat{Un} = id \cdot i_2 \cdot i_2 \cdot i_2 \end{array} \right. \end{array} \right. \end{array} \right. \\
\equiv & \quad \{ \text{ Igualdade extensional, Natural-id } \} \\
 & \left\{ \begin{array}{l} (outExpAr \cdot \underline{X})\ () = i_1\ () \\ \left\{ \begin{array}{l} (outExpAr \cdot N)\ a = (i_2 \cdot i_1)\ a \\ \left\{ \begin{array}{l} (outExpAr \cdot bin)\ (op, (l, r)) = (i_2 \cdot i_2 \cdot i_1)\ (op, (l, r)) \\ (outExpAr \cdot \widehat{Un})\ (op, a) = (i_2 \cdot i_2 \cdot i_2)\ (op, a) \end{array} \right. \end{array} \right. \end{array} \right. \\
\equiv & \quad \{ \text{ Def-comp, Def-const, Def-}N\text{, Def-}bin\text{, Def-Uncurry, Def-}\widehat{Un} \} \\
 & \left\{ \begin{array}{l} outExpAr\ X = i_1\ () \\ \left\{ \begin{array}{l} outExpAr\ (N\ a) = i_2\ \$\ i_1\ a \\ \left\{ \begin{array}{l} outExpAr\ (Bin\ op\ l\ r) = i_2\ \$\ i_2\ \$\ i_1\ (op, (l, r)) \\ outExpAr\ (Un\ op\ a) = i_2\ \$\ i_2\ \$\ i_2\ (op, a) \end{array} \right. \end{array} \right. \end{array} \right.
\end{array}
$$

$\square$

$$
\begin{tikzcd}
ExpAr\ A \arrow[r, bend left, "outExpAr"] \arrow[r, phantom, "\cong"] & 1 + A + BinOp \times (ExpArA)^2 + UnOp \times (ExpArA) \arrow[l, bend left, "inExpAr"]
\end{tikzcd}
$$

$$
\begin{array}{ll}
 & recExprAr\ f = id + (id + (id \times (f \times f) + id \times f)) \\
\equiv & \quad \{ \text{ Def-baseExpAr } \} \\
 & recExprAr\ f = baseExpAr\ id\ id\ id\ f\ f\ id\ f
\end{array}
$$

$\square$

$$
\begin{tikzcd}
ExpAr\ A \arrow[r, bend left, "outExpAr"] \arrow[d, "(\!|g\_eval\ a|\!)"'] & 1 + A + BinOp \times (ExpArA)^2 + UnOp \times (ExpArA) \arrow[l, bend left, "inExpAr"] \arrow[d, "recExpAr\ \ (\!|g\_eval\ a|\!)"] \\
\mathbb{N}_0 & 1 + \mathbb{N}_0 + BinOp \times \mathbb{N}_0^2 + UnOp \times \mathbb{N}_0 \arrow[l, "g\_eval\ \ a"']
\end{tikzcd}
$$

$$
\begin{tikzcd}
ExpAr\ A \arrow[r, bend left, "outExpAr"] & 1 + A + BinOp \times (ExpArA)^2 + UnOp \times (ExpArA) \arrow[l, bend left, "inExpAr"] \\
ExpAr\ A \arrow[u, "[\!(clean)\!]"] \arrow[r, "clean"] & 1 + A + BinOp \times (ExpArA)^2 + UnOp \times (ExpArA) \arrow[u, "recExpAr\ \ [\!(clean)\!]"']
\end{tikzcd}
$$

$$
\begin{tikzcd}
ExpAr\ A \arrow[r, "clean"] \arrow[d, "[\!(clean)\!]"'] & 1 + A + BinOp \times (ExpArA)^2 + UnOp \times (ExpArA) \arrow[d, "recExpAr\ \ [\!(clean)\!]"] \\
ExpAr\ A \arrow[r, bend left, "outExpAr"] \arrow[d, "(\!|gopt\ a|\!)"'] & 1 + A + BinOp \times (ExpArA)^2 + UnOp \times (ExpArA) \arrow[l, bend left, "inExpAr"] \arrow[d, "recExpAr\ \ (\!|gopt\ a|\!)"] \\
\mathbb{N}_0 & 1 + \mathbb{N}_0 + BinOp \times \mathbb{N}_0^2 + UnOp \times \mathbb{N}_0 \arrow[l, "gopt\ \ a"']
\end{tikzcd}
$$

$$
\begin{tikzcd}
ExpAr\ A \arrow[r, bend left, "outExpAr"] \arrow[d, "(\!|sd\_gen|\!)"'] & 1 + A + BinOp \times (ExpArA)^2 + UnOp \times (ExpArA) \arrow[l, bend left, "inExpAr"] \arrow[d, "recExpAr\ (\!|sd\_gen|\!)"] \\
(ExpArA)^2 & 1 + A + BinOp \times ((ExpArA)^2 \times (ExpArA)^2) + UnOp \times (ExpArA)^2 \arrow[l, "sd\_gen"']
\end{tikzcd}
$$

$$\begin{tikzcd}
\mathit{ExpAr}\ A \arrow[rr, bend left, "\mathit{outExpAr}"] \arrow[d, "(\!|\mathit{ad_gen}\ a|\!)"'] & & 1 + A + \mathit{BinOp} \times (\mathit{ExpAr} A)^2 + \mathit{UnOp} \times (\mathit{ExpAr} A) \arrow[ll, bend left, "\mathit{inExpAr}"] \arrow[d, "\mathit{recExpAr}\ (\!|\mathit{ad_gen}\ a|\!)"] \\
\mathbb{N}_0^2 & & 1 + A + \mathit{BinOp} \times (\mathbb{N}_0^2 \times \mathbb{N}_0^2) + \mathit{UnOp} \times (\mathbb{N}_0^2) \arrow[ll, "\mathit{ad_gen}\ \ a"']
\end{tikzcd}$$

Definir:

$$\begin{array}{l}
\mathit{outExpAr}\ X = i_1\ ()\\
\mathit{outExpAr}\ (N\ a) = i_2\ \$\ i_1\ a\\
\mathit{outExpAr}\ (\mathit{Bin}\ \mathit{op}\ l\ r) = i_2\ \$\ i_2\ \$\ i_1\ (\mathit{op}, (l, r))\\
\mathit{outExpAr}\ (\mathit{Un}\ \mathit{op}\ a) = i_2\ \$\ i_2\ \$\ i_2\ (\mathit{op}, a)\\
\mathit{recExpAr}\ f = \mathit{baseExpAr}\ \mathit{id}\ \mathit{id}\ \mathit{id}\ f\ f\ \mathit{id}\ f\\
\mathit{g_eval_exp}\ x\ (i_1\ ()) = x\\
\mathit{g_eval_exp}\ x\ (i_2\ (i_1\ a)) = a\\
\mathit{g_eval_exp}\ x\ (i_2\ (i_2\ (i_1\ (\mathit{Sum}, (e, d))))) = e + d\\
\mathit{g_eval_exp}\ x\ (i_2\ (i_2\ (i_1\ (\mathit{Product}, (e, d))))) = e * d\\
\mathit{g_eval_exp}\ x\ (i_2\ (i_2\ (i_2\ (\mathit{Negate}, a)))) = \mathit{negate}\ a\\
\mathit{g_eval_exp}\ x\ (i_2\ (i_2\ (i_2\ (E, a)))) = \mathit{expd}\ a\\
\mathit{clean}\ a = (\mathit{outExpAr} \cdot h)\ a\ \textbf{where}\\
\quad h\ (\mathit{Bin}\ \mathit{Product}\ (N\,0)\ r) = N\,0\\
\quad h\ (\mathit{Bin}\ \mathit{Product}\ r\ (N\,0)) = N\,0\\
\quad h\ (\mathit{Un}\ E\ (N\,0)) = N\,1\\
\quad h\ (\mathit{Un}\ \mathit{Negate}\ (N\,0)) = N\,0\\
\quad h\ x = x\\
\mathit{gopt}\ a = \mathit{g_eval_exp}\ a
\end{array}$$

$$\begin{array}{l}
\mathit{sd_gen} :: \mathit{Floating}\ a \Rightarrow\\
\quad () + (a + ((\mathit{BinOp}, ((\mathit{ExpAr}\ a, \mathit{ExpAr}\ a),\\
\quad (\mathit{ExpAr}\ a, \mathit{ExpAr}\ a))) + \cdot(\mathit{UnOp}, (\mathit{ExpAr}\ a, \mathit{ExpAr}\ a)))) \rightarrow (\mathit{ExpAr}\ a, \mathit{ExpAr}\ a)\\
\mathit{sd_gen}\ (i_1\ ()) = (X, N\,1)\\
\mathit{sd_gen}\ (i_2\ (i_1\ a)) = (N\ a, N\,0)\\
\mathit{sd_gen}\ (i_2\ (i_2\ (i_1\ (\mathit{Sum}, ((e_1, \mathit{d1}), (e_2, \mathit{d2})))))) = (\mathit{Bin}\ \mathit{Sum}\ e_1\ e_2, \mathit{Bin}\ \mathit{Sum}\ \mathit{d1}\ \mathit{d2})\\
\mathit{sd_gen}\ (i_2\ (i_2\ (i_1\ (\mathit{Product}, ((e_1, \mathit{d1}), (e_2, \mathit{d2}))))))\\
\quad = (\mathit{Bin}\ \mathit{Product}\ e_1\ e_2, \mathit{Bin}\ \mathit{Sum}\ (\mathit{Bin}\ \mathit{Product}\ e_1\ \mathit{d2})\ (\mathit{Bin}\ \mathit{Product}\ \mathit{d1}\ e_2))\\
\mathit{sd_gen}\ (i_2\ (i_2\ (i_2\ (\mathit{Negate}, (e, d))))) = (\mathit{Un}\ \mathit{Negate}\ e, \mathit{Un}\ \mathit{Negate}\ d)\\
\mathit{sd_gen}\ (i_2\ (i_2\ (i_2\ (E, (e, d))))) = (\mathit{Un}\ E\ e, \mathit{Bin}\ \mathit{Product}\ (\mathit{Un}\ E\ e)\ d)
\end{array}$$

$$\begin{array}{l}
\mathit{ad_gen}\ x\ (i_1\ ()) = (x, 1)\\
\mathit{ad_gen}\ x\ (i_2\ (i_1\ a)) = (a, 0)\\
\mathit{ad_gen}\ x\ (i_2\ (i_2\ (i_1\ (\mathit{Sum}, ((e_1, \mathit{d1}), (e_2, \mathit{d2})))))) = (e_1 + e_2, \mathit{d1} + \mathit{d2})\\
\mathit{ad_gen}\ x\ (i_2\ (i_2\ (i_1\ (\mathit{Product}, ((e_1, \mathit{d1}), (e_2, \mathit{d2})))))) = (e_1 * e_2, e_1 * \mathit{d2} + e_2 * \mathit{d1})\\
\mathit{ad_gen}\ x\ (i_2\ (i_2\ (i_2\ (\mathit{Negate}, (e, d))))) = (\mathit{negate}\ e, \mathit{negate}\ d)\\
\mathit{ad_gen}\ x\ (i_2\ (i_2\ (i_2\ (E, (e, d))))) = (\mathit{expd}\ e, d * (\mathit{expd}\ e))
\end{array}$$

Problema 2

Definir

$$
\begin{array}{l}
loop = g \ \mathbf{where} \ g \ (c, a, b) = (c * a \div b, a + 4, b + 1) \\
inic = (1, 2, 2) \\
prj = p \ \mathbf{where} \ p \ (c, \_, \_) = c
\end{array}
$$

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.

$$C_n = \frac{(2n)!}{(n+1)!(n!)} \tag{4}$$

$$
\begin{array}{rcl}
C_0 & = & 1 \\
C_{n+1} & = & \frac{C_n a_n}{b_n} \\
\\
a_n & = & 4n + 2 \\
b_n & = & n + 2 \\
\\
a_0 & = & 2 \\
a_{n+1} & = & a_n + 4 \\
\\
b_0 & = & 2 \\
b_{n+1} & = & b_n + 1
\end{array}
$$

Problema 3

$$
\begin{tikzcd}
\mathbb{Q}^* \arrow[rr, bend left, "outList"] \arrow[d, "(\!|h|\!)_{[\,]}"'] & & 1 + \mathbb{Q} \times \mathbb{Q}^* \arrow[ll, bend left, "inList"] \arrow[d, "id + id \times (\!|h|\!)_{[\,]}"] \\
OverTime\ \mathbb{Q}^* & & 1 + \mathbb{Q} \times OverTime\ \mathbb{Q}^* \arrow[ll, "h = [f,g]"']
\end{tikzcd}
$$

$$
\begin{tikzcd}
{[\mathbb{Q}^*]} \arrow[rr, "coalg"] \arrow[d, "{[\!(coalg)\!]_{[\,]}}"'] & & {[\mathbb{Q}^*] + [\mathbb{Q}^*] \times [\mathbb{Q}^*]} \arrow[d, "{id + [\!(coalg)\!]_{[\,]} \times [\!(coalg)\!]_{[\,]}}"] \\
LTree\ [\mathbb{Q}^*] \arrow[rr, bend left, "out_{\mathsf{LTree}}"] \arrow[d, "(\!|alg|\!)_{\mathsf{LTree}}"'] & & {[\mathbb{Q}^*] + LTree\ [\mathbb{Q}^*] \times LTree\ [\mathbb{Q}^*]} \arrow[ll, bend left, "in_{\mathsf{LTree}}"] \arrow[d, "id + (\!|alg|\!)_{\mathsf{LTree}} \times (\!|alg|\!)_{\mathsf{LTree}}"] \\
OverTime\ \mathbb{Q}^* & & \mathbb{Q}^* + OverTime\ \mathbb{Q}^* \times OverTime\ \mathbb{Q}^* \arrow[ll, "alg = {[a,b]}"']
\end{tikzcd}
$$

$$
\begin{array}{l}
calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint) \\
calcLine = (\!|h|\!)_{[\,]} \ \mathbf{where}
\end{array}
$$

$h = [f, g]$ **where**
$\quad f\ \_\ \_ = nil$
$\quad g\ \_\ [\,] = nil$
$\quad g\ (d, f)\ (x : xs) = \lambda z \rightarrow concat\ \$\ (sequenceA\ [singl \cdot linear1d\ d\ x, f\ xs])\ z$

$deCasteljau :: [NPoint] \rightarrow OverTime\ NPoint$
$deCasteljau = hyloAlgForm\ alg\ coalg$ **where**
$\quad coalg = c$ **where**
$\quad\quad c\ [\,] = i_1\ [\,]$
$\quad\quad c\ [a] = i_1\ [a]$
$\quad\quad c\ l = i_2\ (init\ l, tail\ l)$
$\quad alg = [a, b]$ **where**
$\quad\quad a\ [\,] = nil$
$\quad\quad a\ [x] = \underline{x}$
$\quad\quad b\ (e, d) = \lambda pt \rightarrow (calcLine\ (e\ pt)\ (d\ pt))\ pt$
$hyloAlgForm = hyloLTree$

Uma outra solução para o deCasteljau, criando um novo tipo de dados intermedio.

$deCasteljau' :: [NPoint] \rightarrow OverTime\ NPoint$
$deCasteljau' = hyloAlgForm'\ alg\ coalg$ **where**
$\quad coalg = (id + (id + \langle init, tail \rangle)) \cdot outSL$
$\quad alg = [\underline{nil}, a]$
$\quad a = [\underline{\cdot}, b]$
$\quad b\ (e, d) = \lambda pt \rightarrow (calcLine\ (e\ pt)\ (d\ pt))\ pt$
$outSL\ [\,] = i_1\ ()$
$outSL\ [a] = i_2\ (i_1\ a)$
$outSL\ l = i_2\ (i_2\ l)$
$hyloAlgForm' = h$ **where**
$\quad h\ a\ b = (\!|a|\!)_{\mathsf{Castel}} \cdot [\!(b)\!]_{\mathsf{Castel}}$

data $\mathsf{Castel}\ a = Empty \mid Single\ a \mid InitTail\ (\mathsf{Castel}\ a, \mathsf{Castel}\ a)$ **deriving** $Show$

$in_{\mathsf{Castel}} = [\underline{Empty}, [Single, InitTail]]$

$out_{\mathsf{Castel}}\ Empty = i_1\ ()$
$out_{\mathsf{Castel}}\ (Single\ a) = i_2\ (i_1\ a)$
$out_{\mathsf{Castel}}\ (InitTail\ (e, d)) = i_2\ (i_2\ (e, d))$

$rec_{\mathsf{Castel}}\ f = id + (id + f \times f)$

$(\!|f|\!)_{\mathsf{Castel}} = f \cdot rec_{\mathsf{Castel}}\ (\!|f|\!)_{\mathsf{Castel}} \cdot out_{\mathsf{Castel}}$
$[\!(g)\!]_{\mathsf{Castel}} = in_{\mathsf{Castel}} \cdot rec_{\mathsf{Castel}}\ [\!(g)\!]_{\mathsf{Castel}} \cdot g$

Problema 4

$$
\begin{tikzcd}
\mathbb{N}_0^* \arrow[rr, bend left, "out_{\mathsf{NList}}"] \arrow[d, "(\![b,q]\!)_{\mathsf{NList}} = avg\_aux"'] & & \mathbb{N}_0 + \mathbb{N}_0 \times \mathbb{N}_0^* \arrow[ll, bend left, "in_{\mathsf{NList}}"] \arrow[d, "id + id \times (\![b,q]\!)_{\mathsf{NList}}"] \\
\mathbb{N}_0 \times \mathbb{N} & & \mathbb{N}_0 + \mathbb{N}_0 \times (\mathbb{N}_0 \times \mathbb{N}) \arrow[ll, "{[b,q]}"']
\end{tikzcd}
$$

$$\langle avg, length \rangle = (\![[b, q]]\!)_{\mathsf{NList}}$$

$$\equiv \quad \{ \text{ Universal-cata } \}$$

$$\langle avg, length \rangle \cdot in_{\mathsf{NList}} = [b, q] \cdot rec_{\mathsf{NList}}\ \langle avg, length \rangle$$

$$\equiv \quad \{ \text{ Fusão-+, Absorção-+, Eq-+, Definição de } in_{\mathsf{NList}}\text{, Definição de } rec_{\mathsf{NList}} \}$$

$$\left\{ \begin{array}{l} \langle avg, length \rangle \cdot singl = b \cdot id \\ \langle avg, length \rangle \cdot cons = q \cdot id \times \langle avg, length \rangle \end{array} \right.$$

$$\equiv \quad \{ \text{ Igualdade extensional, Natural-id } \}$$

$$\left\{ \begin{array}{l} (\langle avg, length \rangle \cdot singl)\ x = b\ x \\ (\langle avg, length \rangle \cdot cons)\ (x, xs) = (q \cdot id \times \langle avg, length \rangle)\ (x, xs) \end{array} \right.$$

$$\equiv \quad \{ \text{ Def-comp, Natural-id, Def-}\times\text{, Def-split, Definição de singl, Definição de cons } \}$$

$$\left\{ \begin{array}{l} \langle avg, length \rangle\ [x] = b\ x \\ \langle avg, length \rangle\ (x : xs) = q\ (x, (avg\ xs, length\ xs)) \end{array} \right.$$

$\square$

Solução para listas não vazias:

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

$$\begin{array}{l}
in_{\mathsf{NList}} = [singl, cons] \\
out_{\mathsf{NList}}\ [a] = i_1\ a \\
out_{\mathsf{NList}}\ (a : x) = i_2\ (a, x) \\
rec_{\mathsf{NList}}\ f = id + id \times f \\
(\!|g|\!)_{\mathsf{NList}} = g \cdot rec_{\mathsf{NList}}\ (\!|g|\!)_{\mathsf{NList}} \cdot out_{\mathsf{NList}} \\
avg_aux = (\![[b, q]]\!)_{\mathsf{NList}}\ \mathbf{where} \\
\quad b\ x = (x, 1) \\
\quad q\ (x, (a, l)) = ((x + (a * l))\ /\ (l + 1), l + 1)
\end{array}$$

$$\begin{tikzcd}
LTree\ \mathbb{N}_0 \arrow[rr, bend left, "out_{\mathsf{LTree}}"] \arrow[d, "(\!|gene|\!)_{\mathsf{LTree}}"'] & & \mathbb{N}_0 + LTree\ \mathbb{N}_0 \times LTree\ \mathbb{N}_0 \arrow[ll, bend left, "in_{\mathsf{LTree}}"] \arrow[d, "id + (\!|gene|\!)_{\mathsf{LTree}} \times (\!|gene|\!)_{\mathsf{LTree}}"] \\
\mathbb{N}_0 \times \mathbb{N} & & \mathbb{N}_0 + (\mathbb{N}_0 \times \mathbb{N}) \times (\mathbb{N}_0 \times \mathbb{N}) \arrow[ll, "gene\ =\ [b,q]"']
\end{tikzcd}$$

$$\langle avg, length \rangle = (\!|gene|\!)_{\mathsf{LTree}}$$

$$\equiv \quad \{ \text{ Universal-cata, gene} = [b, q] \ \}$$

$$\langle avg, length \rangle \cdot in_{\mathsf{LTree}} = [b, q] \cdot recLTree\ \langle avg, length \rangle$$

$$\equiv \quad \{ \text{ Fusão-+, Absorção-+, Eq-+, Definição de inL, Definição de recLTree } \}$$

$$\left\{ \begin{array}{l} \langle avg, length \rangle \cdot Leaf = b \cdot id \\ \langle avg, length \rangle \cdot Fork = q \cdot \langle avg, length \rangle \times \langle avg, length \rangle \end{array} \right.$$

$$\equiv \quad \{ \text{ Igualdade extensional, Natural-id } \}$$

$$\left\{ \begin{array}{l} (\langle avg, length \rangle \cdot Leaf)\ a = b\ a \\ (\langle avg, length \rangle \cdot Fork)\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a) = (q \cdot \langle avg, length \rangle \times \langle avg, length \rangle)\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a) \end{array} \right.$$

$$\equiv \quad \{ \text{ Def-comp, Natural-id, Def-}\times\text{, Def-split, Definição de Leaf, Definição de Fork } \}$$

$$\begin{cases} \langle avg, length \rangle\ (Leaf\ a) = b\ a \\ \langle avg, length \rangle\ (Fork\ (\mathsf{LTree}\ a, \mathsf{LTree}\ a)) = q\ ((avg\ (\mathsf{LTree}\ a), length\ (\mathsf{LTree}\ a)), (avg\ (\mathsf{LTree}\ a), length\ (\mathsf{LTree}\ a))) \end{cases}$$
$\square$

Solução para árvores de tipo LTree:

$$\begin{array}{l} avgLTree = \pi_1 \cdot (\!|gene|\!)_{\mathsf{LTree}}\ \mathbf{where} \\ \quad gene = [b, q] \\ \quad b\ a = (a, 1) \\ \quad q\ ((a1, l1), (a2, l2)) = (((a1 * l1) + (a2 * l2))\ /\ (l1 + l2), l1 + l2) \end{array}$$

Problema 5

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

E Resoluções Alternativas e Simplificações sugeridas por Alef

Nessa seção mostro uma forma alternativa que percebi de resolver alguns problemas que não poderiam ser colocadas na seção principal por alguma razão, entre elas por usarem extensões que não estavam já no trabalho.

Também tem simplificações triviais que acho que, para alguns pode facilitar o entendimento. São simplesmente funções que já estão no trabalho reescritas com uma sintaxe mais simples em Haskell (muitas vezes usando lambda case). Conversando com o Tiago, ele achou que seria mais complicado explicar dessa forma. Como no final são equivalentes, decidi deixar aqui caso alguém ache mais fácil ou simplesmente esteja interessado em ver um pouco mais da sintaxe de Haskell, já que a última vez que tivemos uma cadeira que usasse a linguagem diretamente foi no 1° ano.

E.1 Problema 1

Para criar uma interpretação de um tipo A como um tipo B. Assim, por exemplo, posso definir que uma expressão x do tipo $\mathsf{ExpAr}\ a$ pode ser interpretada como $i_1\ ()$ do tipo $OutExpAr\ a$.

$$\begin{array}{l} \mathbf{class}\ Interpretation\ a\ b\ \mathbf{where} \\ \quad to :: a \to b \end{array}$$

A fim de diminuir o número de parêntese e facilitar a legibilidade defini as funções:
bimap de tuplos $((,))$:

$$\begin{array}{l} \mathbf{infixr}\ 6 \times \\ \mathbf{type}\ a \times b = (a, b) \\ (\times) :: (a \to b) \to (c \to d) \to (a, c) \to (b, d) \\ (\times) = (\times) \end{array}$$

bimap de $\cdot + \cdot$:

$$\begin{array}{l} \mathbf{infixr}\ 4 + \\ (+) :: (a \to b) \to (c \to d) \to a \coprod c \to b \coprod d \\ (+) = (+) \end{array}$$

$$\begin{array}{l} \mathbf{infixr}\ 4 \coprod \\ \mathbf{type}\ (\coprod) = \cdot + \cdot \\ (\coprod) :: (a \to c) \to (b \to c) \to a \coprod b \to c \\ (\coprod) = [\cdot, \cdot] \end{array}$$

Novamente, para simplificar a tipagem:

$$\mathbf{type}\ BinExp\ d = BinOp \times \mathsf{ExpAr}\ d \times \mathsf{ExpAr}\ d$$

Note que por conta de precedência $BinExp\ d \equiv BinOp \times (\mathsf{ExpAr}\ d \times \mathsf{ExpAr}\ d)$.

$$\textbf{type}\ UnExp\ d = UnOp \times \mathsf{ExpAr}\ d$$

Isso é uma redefinição do que o Professor definiu. É igual excepto os símbolos mais fáceis de ler.

$$\textbf{type}\ OutExpAr\ a = () \coprod a \coprod BinExp\ a \coprod UnExp\ a$$

Vamos criar uma interpretação de $\mathsf{ExpAr}\ a$ como $OutExpAr\ a$. Ou seja, essa interpretação é out_{ExpAr}.

$$\begin{array}{l}
\textbf{instance}\ Interpretation\ (\mathsf{ExpAr}\ a)\ (OutExpAr\ a)\ \textbf{where} \\
\quad to\ X = i_1\ () \\
\quad to\ (N\ a) = i_2\ \$\ i_1\ a \\
\quad to\ (Bin\ op\ l\ r) = i_2\ \$\ i_2\ \$\ i_1\ (op, (l, r)) \\
\quad to\ (Un\ op\ a) = i_2\ \$\ i_2\ \$\ i_2\ (op, a)
\end{array}$$

Como dito, temos

$$\begin{array}{l}
out_{\mathsf{ExpAr}} :: \mathsf{ExpAr}\ a \to OutExpAr\ a \\
out_{\mathsf{ExpAr}} = to :: \mathsf{ExpAr}\ a \to OutExpAr\ a
\end{array}$$

Agora vou interpretar os símbolos que representam as operações como as funções que representam essas operações.

Interpretamos cada símbolo $BinOp$ como uma função $(c, c) \to c$. Por exemplo, o símbolo Sum é interpretado como a função add.

$$\begin{array}{l}
\textbf{instance}\ (Num\ c) \Rightarrow Interpretation\ BinOp\ ((c, c) \to c)\ \textbf{where} \\
\quad to\ Sum = add \\
\quad to\ Product = mul
\end{array}$$

Na nossa linguagem mais usual:

$$\begin{array}{l}
out_{BinOp} :: Num\ c \Rightarrow BinOp \to (c \times c) \to c \\
out_{BinOp} = to :: (Num\ c) \Rightarrow BinOp \to (c \times c \to c)
\end{array}$$

Interpretamos cada símbolo $UnOp$ como uma função $c \to c$ onde c é da classe $Floating$.

$$\begin{array}{l}
\textbf{instance}\ (Floating\ c) \Rightarrow Interpretation\ UnOp\ (c \to c)\ \textbf{where} \\
\quad to\ Negate = negate \\
\quad to\ E = Prelude.exp
\end{array}$$

Na nossa linguagem mais usual

$$\begin{array}{l}
out_{UnOp} :: Floating\ c \Rightarrow UnOp \to (c \to c) \\
out_{UnOp} = to :: (Floating\ c) \Rightarrow UnOp \to (c \to c)
\end{array}$$

Graças a essas funções auxiliares (que acho intuitivas), podemos simplificar a escrita de g_eval_exp:

$$\begin{array}{l}
g_eval_exp :: Floating\ c \Rightarrow c \to b \coprod c \coprod BinOp \times c \times c \coprod UnOp \times c \to c \\
g_eval_exp\ a = \underline{a} \coprod id \coprod ap \cdot (out_{BinOp} \times id) \coprod ap \cdot (out_{UnOp} \times id)
\end{array}$$

Sabemos que $e^0 = 1$, $-0 = 0$ e $a = 0 \lor b = 0 \Longrightarrow ab = 0$. Nós optimizamos esses 4 casos:

$$\begin{array}{l}
clean :: (Eq\ a, Num\ a) \Rightarrow \mathsf{ExpAr}\ a \to OutExpAr\ a \\
clean = \lambda\textbf{case} \\
\quad (Un\ E\ (N\ 0)) \to tag\ 1 \\
\quad (Un\ Negate\ (N\ 0)) \to tag\ 0 \\
\quad (Bin\ Product\ (N\ 0)\ _) \to tag\ 0 \\
\quad (Bin\ Product\ _\ (N\ 0)) \to tag\ 0 \\
\quad a \to out_{\mathsf{ExpAr}}\ a \\
\quad \textbf{where}\ tag = i_2 \cdot i_1
\end{array}$$

Baseado na função *dup* definida em `Cp.hs`:

$$\mathbf{type}\ Dup\ d = d \times d$$

Mais dois sinônimos:

$$\begin{array}{l}\mathbf{type}\ Bin\ d = BinOp \times Dup\ d \\ \mathbf{type}\ Un\ d\ = UnOp \times d\end{array}$$

A fim de criar código mais sucinto extrai o que se repetia nas funções *sd_gen* e *ad_gen* do Tiago.

$$\begin{array}{l}
bin_{aux}\ \ :: (t \to t \to t) \to (t \to t \to t) \to (BinOp, Dup\ (Dup\ t)) \to Dup\ t \\
bin_{aux}\ \diamondsuit\ \Box\ (op, ((e_1, d_1), (e_2, d_2))) = \mathbf{case}\ op\ \mathbf{of} \\
\quad Sum \quad\ \ \to (e_1 \diamondsuit e_2, d_1 \diamondsuit d_2) \\
\quad Product \to (e_1 \Box e_2, (e_1 \Box d_2) \diamondsuit (d_1 \Box e_2)) \\
un_{aux}\ \ :: (a \to b) \to (b \to a \to b) \to (a \to b) \to (UnOp, Dup\ a) \to Dup\ b \\
un_{aux}\ \ominus\ \circledast\ \boxminus\ (op, (e, d)) = \mathbf{case}\ op\ \mathbf{of} \\
\quad Negate \to (\ominus\ e, \ominus\ d) \\
\quad E \qquad\ \ \to (\boxminus\ e, (\boxminus\ e) \circledast d)
\end{array}$$

Agora podemos escrever:

$$\begin{array}{l}
sd_gen :: Floating\ a \Rightarrow () \amalg a \amalg Bin\ (Dup\ (\mathsf{ExpAr}\ a)) \amalg Un\ (Dup\ (\mathsf{ExpAr}\ a)) \to Dup\ (\mathsf{ExpAr}\ a) \\
sd_gen = f \amalg g \amalg h \amalg k\ \mathbf{where} \\
\quad f\ \ \ = \underline{(X, N\ 1)} \\
\quad g\ a = (N\ a, N\ 0) \\
\quad h\ \ \ = bin_{aux}\ (Bin\ Sum)\ (Bin\ Product) \\
\quad k\ \ \ = un_{aux}\ (Un\ Negate)\ (Bin\ Product)\ (Un\ E) \\
ad_gen :: Floating\ a \Rightarrow a \to () \amalg a \amalg (BinOp, Dup\ (Dup\ a)) \amalg (UnOp, Dup\ a) \to Dup\ a \\
ad_gen\ x = f \amalg g \amalg h \amalg k\ \mathbf{where} \\
\quad f\ \ \ = \underline{(x, 1)} \\
\quad g\ a = (a, 0) \\
\quad h\ \ \ = bin_{aux}\ (+)\ (*) \\
\quad k\ \ \ = un_{aux}\ negate\ (*)\ expd
\end{array}$$

E.2 Problema 3

É interessante ver que podemos ver *calcLine* como um hilomorfismo.

A ideia que levou a isso parte da definição alternativa $calcLine = zipWithM\ linear1d$.

Sabemos que:

$$\begin{array}{l}
zipWithM :: (Applicative\ m) \Rightarrow (a \to b \to m\ c) \to [a] \to [b] \to m\ [c] \\
zipWithM\ f\ xs\ ys = sequenceA\ (zipWith\ f\ xs\ ys)
\end{array}$$

Percebi que podia escrever uma função ($\overline{zip}$):

$$\begin{array}{l}
zip' :: [a] \times [b] \to [a \times b] \\
zip' = [\!(out_{A^* \times B^*})\!]_{[\,]}
\end{array}$$

Desde que transforme os pares de lista de uma forma que respeite o funcionamento de *zipWith* que será descrito em seguida dessa definição:

$$\begin{array}{l}
out_{A^* \times B^*} :: [a] \times [b] \to () + ((a \times b) \times ([a] \times [b])) \\
out_{A^* \times B^*} = \lambda\mathbf{case} \\
\quad ([\,], _) \qquad\quad \to i_1\ () \\
\quad (_, [\,]) \qquad\quad \to i_1\ () \\
\quad (a : as, b : bs) \to i_2\ ((a, b), (as, bs))
\end{array}$$

zipWith pega uma função (de aridade 2), por exemplo, f, e duas listas (digamos a e b) e devolve uma lista (digamos c) onde $c[i] = f(a[i], b[i])$ para todo $0 \leq i \leq min\ (length\ a, length\ b)$.

Ora, então posso pegar uma função curried e pegar uma par de listas. Transformo o par de listas numa lista de pares com $out_{A^* \times B^*}$ e aplico a função argumento em cada um dos pares. Logo tenho a seguinte definição:

$$zipWith' :: ((a \times b) \to c) \to ([\,a\,] \times [\,b\,]) \to [\,c\,]$$
$$zipWith'\ f = \mathsf{T}_{[]} f \cdot zip'$$

A próxima etapa é baseada nas seguintes definições

$$sequenceA :: Applicative\ f \Rightarrow t\ (f\ a) \to f\ (t\ a)$$
$$sequenceA = traverse\ id$$
$$traverse :: Applicative\ f \Rightarrow (a \to f\ b) \to t\ a \to f\ (t\ b)$$
$$traverse\ f = sequenceA \cdot \mathsf{T}_{[]} f$$

Ora, vamos ver como *traverse* é definido para listas

$$\textbf{instance}\ Traversable\ [\,]\ \textbf{where}$$
$$\quad traverse\ f = foldr\ cons\_f\ (pure\ [\,])$$
$$\quad\quad \textbf{where}\ cons\_f\ x\ ys = liftA2\ (:)\ (f\ x)\ ys$$

Vou criar um $sequenceA'$ (será uma versão menos genérica de *sequenceA* uma vez que estamos sendo específicos no trabalho com listas).

$$sequenceA = traverse\ id = foldr\ cons\_f\ (pure\ [\,])\ \textbf{where}$$
$$\quad cons\_f\ x\ ys = liftA2\ (:)\ x\ ys$$

Já fizemos o catamorfismo para *foldr* nas aulas:

$$foldrC :: (a \to b \to b) \to b \to [\,a\,] \to b$$
$$foldrC\ f\ u = \llparenthesis [\underline{u}, \widehat{f}] \rrparenthesis_{[\,]}$$

Então temos:

$$sequenceA' :: Applicative\ f \Rightarrow [\,f\ a\,] \to f\ [\,a\,]$$
$$sequenceA' = \llparenthesis [b, \widehat{g}] \rrparenthesis_{[\,]}\ \textbf{where}$$
$$\quad b = \underline{(pure\ [\,])}$$
$$\quad g\ x\ ys = liftA2\ (:)\ x\ ys$$

Sabemos que, em $Applicative\ ((\to)\ r)$, `pure = const` e $liftA2\ q\ f\ g\ x = q\ (f\ x)\ (g\ x)$. Logo:

$$sequenceA' :: [\,a \to b\,] \to a \to [\,b\,]$$
$$sequenceA' = \llparenthesis [b, \widehat{g}] \rrparenthesis_{[\,]}\ \textbf{where}$$
$$\quad b = \underline{\underline{[\,]}}$$
$$\quad g\ x\ ys = (\lambda z \to x\ z : ys\ z)$$

Lembre que $zipWithM'$ como vimos recebia duas listas. No nosso caso essas duas listas (digamos *xs* e *ys*) estão em um só argumento $t = (xs, ys)$ A seguir seguem uma série de equivalências. Dentro de cada `{}` está uma explicação/justificativa do que foi feito de uma passo para outro.

$$zipWithM'\ f\ t = sequenceA'\ (zipWith'\ f\ t)$$
$$\quad \{\ (\cdot)\ f\ g = \lambda x \to f\ (g\ x)\ \}$$
$$zipWithM'\ f \quad = sequenceA' \cdot zipWith'\ f$$
$$\quad \{\ \text{Def-}zipWith'\ \}$$
$$zipWithM'\ f \quad = sequenceA' \cdot (\mathsf{T}_{[]} f \cdot zip')$$
$$\quad \{\ \text{Assoc-comp}\ \}$$
$$zipWithM'\ f \quad = (sequenceA' \cdot \mathsf{T}_{[]} f) \cdot zip'$$
$$\quad \{\ \text{Def-}sequenceA'\ \}$$
$$zipWithM'\ f \quad = (\llparenthesis [\underline{\underline{[\,]}}, \widehat{g}] \rrparenthesis_{[\,]} \cdot \mathsf{T}_{[]} f) \cdot zip'\ \textbf{where}\ g\ x\ ys = (\lambda z \to x\ z : ys\ z)$$

$\{$ Absorção-cata $\}$

$$zipWithM'\ f \ = (\!|[\underline{\underline{[\,]}}, \widehat{g}] \cdot B_{[]}(f, id)|\!)_{[\,]} \cdot zip' \ \textbf{where}\ g\ x\ ys = (\lambda z \rightarrow x\ z : ys\ z)$$

$\{$ Def-baseList $\}$

$$zipWithM'\ f \ = (\!|[\underline{\underline{[\,]}}, \widehat{g}] \cdot (id + f \times id)|\!)_{[\,]} \cdot zip' \ \textbf{where}\ g\ x\ ys = (\lambda z \rightarrow x\ z : ys\ z)$$

$\{$ Absorção-+; Natural-const $\}$

$$zipWithM'\ f \ = (\!|[\underline{\underline{[\,]}}, \widehat{g} \cdot (f \times id)]|\!)_{[\,]} \cdot zip' \ \textbf{where}\ g\ x\ ys = (\lambda z \rightarrow x\ z : ys\ z)$$

$\{$ $(\cdot)\ f\ g = \lambda x \rightarrow f\ (g\ x)$ $\}$

$$zipWithM'\ f \ = (\!|[\underline{\underline{[\,]}}, \lambda(a, b) \rightarrow \widehat{g}\ ((f \times id)\ (a, b))]|\!)_{[\,]} \cdot zip' \ \textbf{where}\ g\ x\ ys = (\lambda z \rightarrow x\ z : ys\ z)$$

$\{$ Def-$\times$ $\}$

$$zipWithM'\ f \ = (\!|[\underline{\underline{[\,]}}, \lambda(a, b) \rightarrow g\ (f\ a, b)]|\!)_{[\,]} \cdot zip' \ \textbf{where}\ g\ (x, ys) = (\lambda z \rightarrow x\ z : ys\ z)$$

$\{$ Deixe que $h = (\lambda(a, b) \rightarrow g\ (f\ a, b))$; Notação-$\lambda$ $\}$

$$zipWithM'\ f \ = (\!|[\underline{\underline{[\,]}}, h]|\!)_{[\,]} \cdot zip' \ \textbf{where}\ h\ (a, b) = (\lambda z \rightarrow (f\ a)\ z : b\ z)$$

$\{$ Def-zip'; Notação-λ $\}$

$$zipWithM'\ f \ = (\!|[\underline{\underline{[\,]}}, h]|\!)_{[\,]} \cdot [\!(out_{A^* \times B^*})\!]_{[\,]} \ \textbf{where}\ h\ (a, b)\ z = (f\ a)\ z : b\ z$$

$\{$ catamorfismo após anamorfismo é um hilomorfismo $\}$

$$zipWithM'\ f \ = [\![[\underline{\underline{[\,]}}, h], out_{A^* \times B^*}]\!]_{[\,]} \ \textbf{where}\ h\ (a, b)\ z = (f\ a)\ z : b\ z$$

$$zipWithM'\ f \ = [\![[\underline{\underline{[\,]}}, h], out_{A^* \times B^*}]\!]_{[\,]} \ \textbf{where}\ h\ (a, b) = cons \cdot \langle f\ a, b \rangle$$

Portanto, lembrando que $calcLine = zipWithM\ linear1d$ e tendo em mente que $calcLine :: [\mathbb{Q}] \rightarrow [\mathbb{Q}] \rightarrow Float \rightarrow [\mathbb{Q}]$, mas $zipWithM' :: ((a \times b) \rightarrow c \rightarrow d) \rightarrow ([a] \times [b]) \rightarrow c \rightarrow [d]$

$$calcLine = \overline{zipWithM'\ \widehat{linear1d}}$$

$\{$ Def-$zipWithM'$ $\}$

$$calcLine = \overline{[\![[\underline{\underline{[\,]}}, h], out_{A^* \times B^*}]\!]_{[\,]}} \ \textbf{where}\ h\ (a, b) = cons \cdot \langle \widehat{linear1d}\ a, b \rangle$$

E.2.1 Notação case

Notação lambda simplifica expressão das funções:

$$\begin{array}{l} outSL :: [a] \rightarrow () \amalg a \amalg [a] \\ outSL = \lambda\textbf{case} \\ \quad [\,] \ \ \rightarrow i_1\ () \\ \quad [a] \rightarrow i_2\ (i_1\ a) \\ \quad l \ \ \ \rightarrow i_2\ (i_2\ l) \end{array}$$

A notação case que acho mais simples mas requer uma extensão não usada no trabalho.

$$\begin{array}{l} out_{\mathsf{Castel}} :: \mathsf{Castel}\ a \rightarrow () \amalg a \amalg \mathsf{Castel}\ a \times \mathsf{Castel}\ a \\ out_{\mathsf{Castel}} = \lambda\textbf{case} \\ \quad Empty \qquad\quad \rightarrow i_1\ () \\ \quad Single\ a \qquad \rightarrow i_2\ (i_1\ a) \\ \quad InitTail\ (e, d) \rightarrow i_2\ (i_2\ (e, d)) \end{array}$$

E.3 Problema 4

E.3.1 Notação case

Notação lambda facilita legibilidade:

$$\begin{array}{l} out_{\mathsf{NList}} :: [a] \rightarrow a \amalg a \times [a] \\ out_{\mathsf{NList}} = \lambda\textbf{case} \\ \quad [a] \rightarrow i_1\ a \\ \quad (a : x) \rightarrow i_2\ (a, x) \end{array}$$

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.