Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2023/24)

Lic. em Ciências da Computação

Grupo G22

a105531 Cláudia Rego Faria
a102502 Patrícia Daniela Fernandes Bastos
a101987 Renato Pereira Garcia

junho 2024

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 e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos 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.

Antes de abordarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao sofware a instalar, etc.

> Valoriza-se a escrita de *pouco* código, que corresponda a soluções simples e elegantes mediante a utilização dos combinadores de ordem superior estudados na disciplina.
> Recomenda-se ainda que o código venha acompanhado de uma descrição de como funciona e foi concebido, apoiado em diagramas explicativos. Para instruções sobre como produzir esses diagramas e exprimir raciocínios de cálculo, ver o anexo D.

Problema 1

No passado dia 10 de Março o país foi a eleições para a Assembleia da República. A lei eleitoral portuguesa segue, como as de muitos outros países, o chamado Método de Hondt para selecionar os candidatos dos vários partidos, conforme os votos que receberam. E, tal como em anos anteriores, há sempre notícias a referir a quantidade de votos desperdiçados por este método. Como e porque é que isso acontece?

Pretende-se nesta questão construir em Hakell um programa que implemente o método de Hondt. A Comissão Nacional de Eleições descreve esse método nesta página, que deverá ser estudada para resolver esta questão. O quadro que aí aparece,

Divisor	Partido			
	A	B	C	D
1	**12000**	**7500**	**4500**	**3000**
2	**6000**	**3750**	2250	1500
3	**4000**	2500	1500	1000
4	3000	1875	1125	750

mostra o exemplo de um círculo eleitoral que tem direito a eleger 7 deputados e onde concorrem às eleições quatro partidos A, B, C e D, cf:

$$\textbf{data}\ Party = A \mid B \mid C \mid D\ \textbf{deriving}\ (Eq, Ord, Show)$$

A votação nesse círculo foi

$$[(A, 12000), (B, 7500), (C, 4500), (D, 3000)]$$

sendo o resultado eleitoral

$$result = [(A,3),(B,2),(C,1),(D,1)]$$

apurado correndo

$$result = final\ history$$

que corresponde à última etapa da iteração:

$$history = [\text{for } step\ db\ i \mid i \leftarrow [0..7]]$$

Verifica-se que, de um total de 27000 votos, foram desperdiçados:

$$wasted = 9250$$

Completem no anexo G as funções que se encontram aí indefinidas[1], podendo adicionar funções auxiliares que sejam convenientes. No anexo F é dado algum código preliminar.

Problema 2

A biblioteca *LTree* inclui o algoritmo "mergesort" (*mSort*), que é um hilomorfismo baseado função

$$merge :: Ord\ a \Rightarrow ([a],[a]) \to [a]$$

que junta duas listas previamente ordenadas numa única lista ordenada.

Nesta questão pretendemos generalizar *merge* a k-listas (ordenadas), para qualquer k finito:

$$mergek :: Ord\ a \Rightarrow [[a]] \to [a]$$

Esta função deverá ser codificada como um hilomorfismo, a saber:

$$mergek = [\![f, g]\!]$$

1. Programe os genes f e g do hilomorfismo *mergek*.
2. Estenda *mSort* a

   $$mSortk :: Ord\ a \Rightarrow Int \to [a] \to [a]$$

   por forma a este hilomorfismo utilizar *mergek* em lugar de *merge* na estapa de "conquista". O que se espera de *mSortk k* é que faça a partição da lista de entrada em k sublistas, sempre que isso for possível. (Que vantagens vê nesta nova versão?)

Problema 3

Considere-se a fórmula que dá o n-ésimo número de Catalan:

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

No anexo F dá-se a função *catdef* que implementa a definição (1) em Haskell. É fácil de verificar que, à medida que n cresce, o tempo que *catdef n* demora a executar degrada-se.

[1] Cf. $\perp$ no código.

Pretende-se uma implementação mais eficiente de C_n que, derivada por recursividade mútua, não calcule factoriais nenhuns:

$$cat = \cdots \text{ for } loop\ init\ \textbf{where}\ \cdots$$

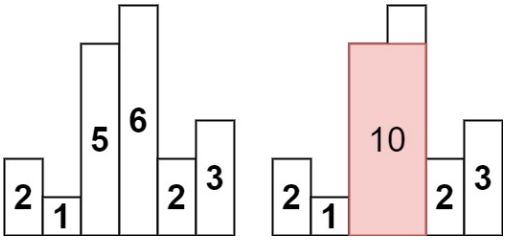
No anexo F é dado um oráculo que pode ajudar a testar *cat*. Deverá ainda ser comparada a eficiência da solução calculada *cat* com a de *catdef*.

> **Sugestão**: Começar por estudar a regra prática que se dá no anexo E para problemas deste género.

Problema 4

Esta questão aborda um problema que é conhecido pela designação *Largest Rectangle in Histogram*. Precebe-se facilmente do que se trata olhando para a parte esquerda da figura abaixo, que mostra o histograma correspondente à sequência numérica:

$$h = [2, 1, 5, 6, 2, 3]$$



À direita da mesma figura identifica-se o rectângulo de maior área que é possível inscrever no referido histograma, com área $10 = 2 * 5$.

Pretende-se a definição de uma função em Haskell

$$lrh :: [Int] \to Int$$

tal que *lrh x* seja a maior área de rectângulos que seja possível inscrever em *x*.

Pretende-se uma solução para o problema que seja simples e estruturada num hilomorfismo baseado num tipo indutivo estudado na disciplina ou definido *on purpose*.

Anexos

A Natureza do trabalho a realizar

Este trabalho teórico-prático deve ser realizado por grupos de 3 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 em **todos** os exercícios do trabalho, para assim poderem responder a qualquer questão colocada na *defesa oral* do relatório.

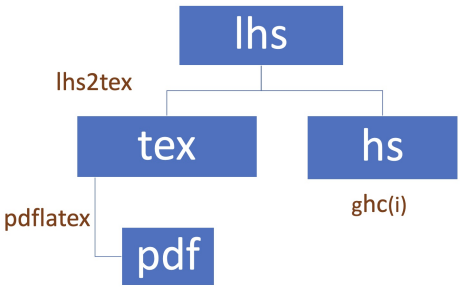
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *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 `cp2324t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2324t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2324t.zip`.

Como se mostra no esquema abaixo, de um único ficheiro (*lhs*) gera-se um PDF ou faz-se a interpretação do código Haskell que ele inclui:



Vê-se assim que, para além do GHCi, serão necessários os executáveis pdflatex e lhs2TeX. Para facilitar a instalação e evitar problemas de versões e conflitos com sistemas operativos, é recomendado o uso do Docker tal como a seguir se descreve.

B Docker

Recomenda-se o uso do container cuja imagem é gerada pelo Docker a partir do ficheiro `Dockerfile` que se encontra na diretoria que resulta de descompactar `cp2324t.zip`. Este container deverá ser usado na execução do GHCi e dos comandos relativos ao LaTeX. (Ver também a `Makefile` que é disponibilizada.)

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

Após instalar o Docker e descarregar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2324t .
$ docker run -v ${PWD}:/cp2324t -it cp2324t
```

NB: O objetivo é que o container seja usado *apenas* para executar o GHCi e os comandos relativos ao LaTeX. Deste modo, é criado um *volume* (cf. a opção `-v ${PWD}:/cp2324t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2324t` no container sejam partilhadas.

O grupo deverá visualizar/editar os ficheiros numa máquina local e compilá-los no container, executando:

```
$ lhs2TeX cp2324t.lhs > cp2324t.tex
$ pdflatex cp2324t
```

lhs2TeX é o pre-processador que faz "pretty printing" de código Haskell em LaTeX e que faz parte já do container. Alternativamente, basta executar

```
$ make
```

para obter o mesmo efeito que acima.

Por outro lado, o mesmo ficheiro `cp2324t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2324t.lhs
```

O grupo deve abrir o ficheiro `cp2324t.lhs` num editor da sua preferência e verificar que assim é: todo o texto que se encontra dentro do ambiente

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

é seleccionado pelo GHCi para ser executado.

C Em que consiste o TP

Em que consiste, então, o *relatório* a que se referiu acima? É a edição do texto que está a ser lido, preenchendo o anexo G 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 cp2324t.aux
$ makeindex cp2324t.idx
```

e recompilar o texto como acima se indicou. (Como já se disse, pode fazê-lo correndo simplesmente `make` no container.)

No anexo F disponibiliza-se algum código Haskell relativo aos problemas que são colocados. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Deve ser feito uso da programação literária para documentar bem o código que se desenvolver, em particular fazendo diagramas explicativos do que foi feito e tal como se explica no anexo D.

> **Importante:** o grupo deve evitar trabalhar fora deste ficheiro lhs que lhe é fornecido. Se, para efeitos de divisão de trabalho, o decidir fazer, deve **regularmente integrar** e validar as soluções que forem sendo obtidas neste lhs, garantindo atempadamente a compatibilidade com este. Se não o fizer corre o risco de vir a submeter um ficheiro que não corre no GHCi e/ou apresenta erros na geração do PDF.

D Como exprimir cálculos e diagramas em LaTeX/lhs2TeX

Como primeiro exemplo, estudar o texto fonte (lhs) do que está a ler[1] onde se obtém o efeito seguinte:[2]

$$
\begin{aligned}
& id = \langle f, g\rangle \\
\equiv & \quad \{\ \text{universal property}\ \} \\
& \begin{cases} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{cases} \\
\equiv & \quad \{\ \text{identity}\ \} \\
& \begin{cases} \pi_1 = f \\ \pi_2 = g \end{cases} \\
\square &
\end{aligned}
$$

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

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

E Regra prática para a recursividade mútua em $\mathbb{N}_0$

Nesta disciplina estudou-se como fazer programação dinâmica por cálculo, recorrendo à lei de recursividade mútua.[3]

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

$$
\begin{aligned}
& fib\ 0 = 1 \\
& fib\ (n+1) = f\ n \\
& f\ 0 = 1 \\
& f\ (n+1) = fib\ n + f\ n
\end{aligned}
$$

[1] Procure e.g. por `"sec:diagramas"`.

[2] Exemplos tirados de [2].

[3] Lei (3.95) em [2], página 110.

Obter-se-á de imediato

$$\begin{aligned}&fib' = \pi_1 \cdot \text{for } loop\ init\ \textbf{where}\\&\quad loop\ (fib, f) = (f, fib + f)\\&\quad init = (1, 1)\end{aligned}$$

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.[1]
- 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[2], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

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

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

$$\begin{aligned}&f'\ a\ b\ c = \pi_1 \cdot \text{for } loop\ init\ \textbf{where}\\&\quad loop\ (f, k) = (f + k, k + 2 * a)\\&\quad init = (c, a + b)\end{aligned}$$

F Código fornecido

Problema 1

Tipos básicos:

$$\begin{aligned}&\textbf{type } Votes = \mathbb{Z}\\&\textbf{type } Deputies = \mathbb{Z}\end{aligned}$$

Dados:

$$\begin{aligned}&db :: [(Party, (Votes, Deputies))]\\&db = \text{map}\ \ f\ vote\ \textbf{where}\ f\ (a, b) = (a, (b, 0))\\&vote = [(A, 12000), (B, 7500), (C, 4500), (D, 3000)]\end{aligned}$$

Apuramento:

$$\begin{aligned}&final = \text{map}\ \ (id \times \pi_2) \cdot last\\&total = sum\ (\text{map}\ \ \pi_2\ vote)\\&wasted = waste\ history\end{aligned}$$

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

[2] Secção 3.17 de [2] e tópico Recursividade mútua nos vídeos de apoio às aulas teóricas.

Problema 3

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[1]:

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

G Soluções dos alunos

Os grupos 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, diagramas e/ou outras funções auxiliares que sejam necessárias.

> **Importante**: Não pode ser alterado o texto deste ficheiro fora deste anexo.

Problema 1

Neste problema, cujo objetivo final era implementar o método de Hondt, primeiramente desenvolvemos as funções *waste*, para resolver a wasted, e a função *step* para descobrir a função history. A função *waste* vai buscar o último elemento da history e a cada elemento da lista aplica a função *restantes*, que calcula o número de votos desperdiçados de um partido. Depois a função soma todos os votos desperdiçados de todos os partidos. Isso vai originar a função *wasted*.

Votos desperdiçados:

$$waste = sum \cdot \text{map}\ \ restantes \cdot last\\ \quad \textbf{where}\ restantes = (/)\ \langle\$\rangle\ fromIntegral \cdot (\pi_1 \cdot \pi_2) <*> fromIntegral \cdot \text{succ}\ \cdot (\pi_2 \cdot \pi_2)$$

A função *step* aplica basicamente a função *update*, que vai atualizar o número de deputados de um partido. A função *update* vai verificar se o partido é o partido com maior quociente e se for, incrementa o número de deputados desse partido. Esta função foi feita baseada no **McCarthy's Conditional**. A função *step* vai ser utilizada para construir a *history*. Para além disso, a variável *maxParty* vai buscar o partido com maior quociente e a função *quotient* vai calcular o quociente de um partido. A variável *pred* vai verificar se o partido é o partido com maior quociente.

Corpo do ciclo-**for**:

$$step\ l = \text{map}\ \ update\ l\\ \quad \textbf{where}\\ \quad\quad quotient\ (\_, (v, d)) = fromIntegral\ v\ /\ fromIntegral\ (\text{succ}\ \ d)\\ \quad\quad maxParty = \pi_1\ \$\ maximumBy\ (comparing\ (quotient))\ l$$

[1] Fonte: Wikipedia.

$$pred = (\equiv maxParty) \cdot \pi_1$$
$$update = (\lambda x \to \textbf{if } pred\ x \textbf{ then } (\lambda(p,(v,d)) \to (p,(v,\text{succ}\ \ d)))\ x \textbf{ else } id\ x)$$

O **History** terá os seguintes passos, no exemplo fornecido neste enunciado:

$$\begin{array}{ll}
i = 0 & [(A,(12000,0)),(B,(7500,0)),(C,(4500,0)),(D,(3000,0))] \\
i = 1 & [(A,(12000,1)),(B,(7500,0)),(C,(4500,0)),(D,(3000,0))] \\
i = 2 & [(A,(12000,1)),(B,(7500,1)),(C,(4500,0)),(D,(3000,0))] \\
i = 3 & [(A,(12000,2)),(B,(7500,1)),(C,(4500,0)),(D,(3000,0))] \\
i = 4 & [(A,(12000,2)),(B,(7500,1)),(C,(4500,1)),(D,(3000,0))] \\
i = 5 & [(A,(12000,3)),(B,(7500,1)),(C,(4500,1)),(D,(3000,0))] \\
i = 6 & [(A,(12000,3)),(B,(7500,2)),(C,(4500,1)),(D,(3000,0))] \\
i = 7 & [(A,(12000,3)),(B,(7500,2)),(C,(4500,1)),(D,(3000,1))]
\end{array}$$

Problema 2

Genes de *mergek*:

O anamorfismo g recebe uma lista de listas ordenadas e cria pares recursivamente. Por sua vez, o catamorfismo f aplica a função merge a pares de listas recursivamente. O resultado do hilomorfismo mergek será uma única lista ordenada.

$$f :: (Ord\ a) \Rightarrow () + ([a],[a]) \to [a]$$
$$f\ (i_1\ ()) = [\,]$$
$$f\ (i_2\ (xs,ys)) = merge\ (xs,ys)$$

$$g :: [[a]] \to () + ([a],[[a]])$$
$$g\ [\,] = i_1\ ()$$
$$g\ [x] = i_2\ (x,[\,])$$
$$g\ (x:xs) = i_2\ (x,xs)$$

O seguinte diagrama representa o hilomorfismo mergek:

$$\begin{array}{ccc}
(\mathbb{N}_0^*)^* & \xrightarrow{\quad g \quad} & 1 + \mathbb{N}_0^* + (\mathbb{N}_0^*)^* \\
{\scriptstyle [\![g]\!]}\downarrow & & \downarrow{\scriptstyle id+id\ x\ g} \\
(\mathbb{N}_0^*)^* & & 1 + \mathbb{N}_0^* + (\mathbb{N}_0^*)^* \\
{\scriptstyle (\!| f |\!)}\downarrow & & \downarrow{\scriptstyle id+id\ x\ f} \\
\mathbb{N}_0^* & \xleftarrow[\quad f \quad]{} & 1 + \mathbb{N}_0^* + \mathbb{N}_0^*
\end{array}$$

Extensão de *mSort*:

Foi criada uma função lsplitk" que parte uma lista em k sublistas recursivamente. Se k receber o valor de 1, é dado um caso de paragem que coloca cada elemento da lista numa lista dentro da nova lista. Este caso foi implementado para não conduzir a um loop infinito. Esta nova função lsplitk" permite criar um hilomorfismo mSortk que implementa mergek onde mSort implementaria merge.

$$mSortk\ k = [singl, mergek] \cdot (id + \text{map}\ \ (mSortk\ k)) \cdot (lsplitk''\ k)$$

```
lsplitk :: Int → [a] → [[a]]
lsplitk k l = lsplitk' k (length l) l
  where
    lsplitk' _ _ [] = []
    lsplitk' 0 _ _ = []
    lsplitk' i n l  = take tamanho l : lsplitk' (i - 1) (n - tamanho) (drop tamanho l)
      where
        tamanho = (n + i - 1) `div` i


lsplitk'' :: Int → [a] → a + [[a]]
lsplitk'' _ [x] = i1 x
lsplitk'' 1 xs = i2 $ map  (λx → [x]) xs
lsplitk'' k xs = i2 (lsplitk k xs)
```

O algoritmo mSort original divide a lista em dois recursivamente até que cada sublista contenha apenas um elemento. A implementação de mSortk, que recebe um inteiro k pelo utilizador, permite ajustar a divisão da lista, restringindo o número de chamadas recursivas ao necessário e resultando num desempenho mais eficiente.

Problema 3

Sendo, $catalan(n) = \frac{(2n)!}{(n+1)!(n!)}$

é imediato que $catalan(0) = 1$.

Para *cat* ser uma implementação mais eficiente de *catalan*, derivada por recursividade mútua, não calculando factoriais nenhuns, é necessário encontrar um *k* tal que,

$$catalan(n+1) = k * catalan(n)$$

Para tal fazemos os seguintes cálculos matemáticos:

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

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

Temos então

$$catalan(n+1) = \frac{4n+2}{(n+2)} * catalan(n)$$

Sendo $k(n) = \frac{4n+2}{(n+2)}$ podemos dividir *k* em duas funções: $num = 4n + 2$ e $denom = n + 2$

Onde, $num(0) = 2$

$num(n+1) = 4(n+1) + 2 = 4n + 4 + 2 = 4 + 4n + 2 = num(n) + 4$

$denom(0) = 2$

$denom(n+1) = n + 1 + 2 = n + 2 + 1 = denom(n) + 1$

Assim, temos:

$$catalan\ 0 = 1$$
$$catalan\ (n+1) = num\ (n) * catalan\ (n)\ `div`\ denom\ (n)$$

$$num\ 0 = 2$$
$$num\ (n+1) = num\ (n) + 4$$

$$denom\ 0 = 2$$
$$denom\ (n+1) = denom\ (n) + 1$$

Podemos concluir que:

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

onde:

$$loop\ (catalan, num, denom) = ((num * catalan)\ `div`\ denom, num + 4, denom + 1)$$
$$inic = (1, 2, 2)$$
$$prj\ (catalan, num, denom) = catalan$$

catdef é menos eficiente que *cat* principalmente quando o n é elevado, como mencionado no enunciado, por requerir múltiplos cálculos de factoriais repetitivos. Já *cat* vai atualizando os valores de *catalan*, *num*, e *denom* a cada passo evitando cálculos repetitivos.

Problema 4

De forma a solucionar o problema de forma simples, *lrh* é definida por um hilomorfismo e pela função *geraListas*:

$$lrh = [\![c, a]\!] \cdot geraListas$$

A função *geraListas* recebe a lista das alturas de cada barra do histograma e gera as várias sublistas que podem estar contidas em tal.

$$geraListas :: [Int] \to [[Int]]$$
$$geraListas\ heights = concatMap\ inits\ (tails\ heights)$$

Após isso é então efetuado o hilomorfismo que recebe a lista de listas gerada por *geraListas*. É pelo anamorfismo *a* que é calculada a área para cada lista recursivamente com o auxilio da função *calculaArea*.

$$a :: [[Int]] \to () + (Int, [[Int]])$$
$$a\ [\,] = i_1\ ()$$
$$a\ (x : xs) = i_2\ (calculaArea\ x, xs)$$

calculaArea calcula a área do retângulo inscrito na lista recebida.

$$calculaArea :: [Int] \to Int$$
$$calculaArea\ [\,] = 0$$
$$calculaArea\ heights = minimum\ heights * length\ heights$$

É pelo catamorfismo c que é calculada, recursivamente, a maior área entre as áreas de todos os retângulos que são possíveis inscrever no histograma.

$$
\begin{aligned}
&c :: () + (Int, Int) \to Int \\
&c\ (i_1\ ()) = 0 \\
&c\ (i_2\ (xs, ys)) = max\ xs\ ys
\end{aligned}
$$

Os seguintes diagramas ilustram *geraListas* e o hilomorfismo respetivamente:

$$\mathbb{N}_0^* \xrightarrow{geraListas} (\mathbb{N}_0^*)^*$$

$$
\begin{array}{ccc}
(\mathbb{N}_0^*)^* & \xrightarrow{a} & 1 + \mathbb{N}_0 + (\mathbb{N}_0^*)^* \\
{\scriptstyle [\![a]\!]}\downarrow & & \downarrow{\scriptstyle id+id\times a} \\
\mathbb{N}_0^* & & 1 + \mathbb{N}_0 + \mathbb{N}_0^* \\
{\scriptstyle (\!| c |\!)}\downarrow & & \downarrow{\scriptstyle id+id\times c} \\
\mathbb{N}_0 & \xleftarrow[c]{} & 1 + \mathbb{N}_0 + \mathbb{N}_0
\end{array}
$$

Index

References

[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.