Cálculo de Programas
Trabalho Teórico-Prático (Parte I)
LCC — Ano Lectivo de 2008/09

J.N. Oliveira & H.D. Macedo
Departamento de Informática
Universidade do Minho

Junho de 2009

Conteúdo

1 Preâmbulo

A disciplina de Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de uma álgebra da programação (conjunto de leis universais e seus corolários) e aplica-se essa álgebra a problemas de programação por resolução de equações cujas incógnitas são os próprios programas que se querem obter.

Por razões de ordem pedagógica, restringe-se a aplicação desta ciência ao desenvolvimento de programas funcionais e usa-se a linguagem HASKELL como veículo de programação [1].

O presente trabalho tem por objectivo concretizar na prática os objectivos da disciplina acima enunciados: (a) por um lado, colocar os alunos perante problemas de programação que deverão ser resolvidos em HASKELL; (b) por outro lado, convidar os alunos a documentarem esses mesmos programas com os cálculos que precederam a sua codificação. Há ainda um terceiro objectivo: o de ensinar a produzir textos técnico-científicos de qualidade.

Para cumprir de forma integrada e simples os objectivos acima vamos recorrer a uma técnica de programação dita *literária* [3], cujo princípio base é o seguinte: *a documentação de um programa deve coincidir com ele próprio*. Por outras palavras, o código fonte e a sua documentação deverão constar do mesmo documento (ficheiro). O texto que está a ler é um exemplo de *programação literária*, como adiante se referirá.

2 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos com um máximo de três alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que constam da página da disciplina na *internet*, onde também se encontra o ficheiro original do presente texto. Recomenda-se uma abordagem equilibrada e 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? Deverá ser um texto de programação literária [3] que responda aos requisitos e questões de programação que constam deste enunciado, mais adiante.

3 Como documentar código HASKELL

Na programação científica, como é aquela que se ensina nesta disciplina, é tão importante produzir um bom programa, ou pacote de programas (vulg. *aplicação de 'software'*) como produzir evidência sobre a forma como foi concebido (decisões tomadas, etc). Este aspecto faz com que a *documentação* sobre um programa seja tanto ou mais valiosa que o programa propriamente dito. (Essa documentação é, afinal, o mais precioso e duradouro bem de que uma empresa dispõe sempre que um autor de um produto de 'software' seu a abandona.)

Um erro clássico dos hábitos tradicionais de programação é o de se escrever a documentação só depois do respectivo programa *estar pronto*. Por via de regra,

[1] A álgebra da programação estende-se à lógica em geral, encarada como liguagem para especificar problemas e construir soluções. Esta continuação desta disciplina aborda-se na unidade curricular *Métodos Formais em Engenharia de Software* do Mestado em Informática desta Universidade, cf.

`http://www.di.uminho.pt/ensino/formacao-avancada/metodos-formais-em-engenharia-de-software`.

a documentação escrita nessa fase tardia é omissa e de má qualidade. Isto porque muitos detalhes do desenvolvimento já estão esquecidos nessa altura, sendo muitas vezes tudo escrito *a correr*.

Todos são unânimes em recomendar que a documentação seja escrita *ao mesmo tempo que* o código, e para isso nada melhor do que usar o *mesmo ficheiro* de texto para registar código e documentação. Como já se disse acima, chama-se a esta estratégia programação literária, sendo o HASKELL uma das linguagens em que tal estratégia é prática comum.

É o que se passa com o texto (PDF) que neste momento está a ler: foi gerado a partir de um ficheiro (`cp0809t.lhs`) usando o processador de texto LaTeX [2]; mas, se submeter o mesmo ficheiro ao seu interpretador de HASKELL preferido, verá que o código nele incluído será seleccionado, carregado e executado.

Exemplo: vamos declarar de seguida os isomorfismos in e out associados à definição de listas em HASKELL tal como foram definidos nas aulas desta disciplina. Primeiro vamos dar um nome ao módulo que estamos a definir

```
module Cp0809t where
import Cp
import Nat
import System.Time
import Graphics.SOE
```

aproveitando para indicar que ele depende de funções disponíveis nos módulos Cp, Nat, etc. Basta, então escrevermos:

$$inl = [\underline{[]}, \widehat{(:)}]$$

$$out\ [\,] = i_1\ ()$$
$$out\ (a : x) = i_2\ (a, x)$$

Se se inspeccionar o ficheiro `cp0809t.lhs` vê-se que todo o texto indentado acima está entre dois delimitadores, `\begin{code}` e `\end{code}`. É nisso que consiste o protocolo que o LaTeX e o HASKELL estabelecem entre si: quando o LaTeX encontra blocos `\begin{code} ... \end{code}`, limita-se a mostrá-los em formato *pretty-printed*. Por seu lado, o HASKELL está preparado para ignorar tudo o que é texto LaTeX *e só prestar atenção* ao que está dentro desses blocos.

Por exemplo, se se submeter `cp0809t.lhs` ao GHCi [2], ter-se-á evidência de que tudo correu bem,

```
GHCi, version 6.10.1: http://www.haskell.org/ghc/
Loading package ghc-prim ... linking ... done.
Loading package integer ... linking ... done.
Loading package base ... linking ... done.
[1 of 3] Compiling Cp               ( Cp.hs, interpreted )
[2 of 3] Compiling Nat              ( Nat.hs, interpreted )
[3 of 3] Compiling Cp0809t          ( cp0809t.lhs, interpreted )
Ok, modules loaded: Cp0809t, Nat, Cp.
```

podendo-se de imediato questionar o interpretador.

Por exemplo, pode pedir-se para este calcular out `["a","b","c"]`, devendo-se obter

```
*Cp0809t> out ["a","b","c"]
Right ("a",["b","c"])
```

[2]Note que tem de ter acessível os ficheiros `Cp.hs`, etc que são importados. Note ainda que deve invocar o GHCi com as opções de comando `-XMultiParamTypeClasses`.

etc.

Continuando, vamos agora declarar o combinador $(\!|\_|\!)$ que nos permite construir *catamorfismos* de listas:

$$cata\ g = g \cdot rec\ (cata\ g) \cdot out$$

onde

$$rec\ f = id + id \times f$$

Voltando de novo ao GHCi, podemos inquirir qual o tipo de $cata$,

```
*Cp0809t> :t cata
cata :: forall c a. (Either () (a, c) -> c) -> [a] -> c
```

qual o resultado da aplicação do catamorfismo

$$(\!|[\underline{0}, (1+) \cdot \pi_2]|\!) \tag{1}$$

à lista acima,

```
*Cp0809t> cata (either (const 0) ((1+) . p2)) ["a","b","c"]
3
```

etc.

4 Como documentar cálculos de programas

Nas aulas desta disciplina, sempre que se pretende derivar um programa em HASKELL, ou raciocinar sobre ele, não se usa a notação HASKELL directamente, mas sim uma notação mais compacta, de feição matemática, que permite abreviar texto a agilizar os cálculos. A expressão (1) acima, por exemplo, é a que é usada no cálculo de catamorfismos e não a expressão HASKELL que acima se passou ao GHCi. Se quisermos, podemos ver (1) como o "pretty-printing" do referido texto em sintaxe HASKELL.

Assim, não vamos usar *programação literária* para documentar os cálculos, mas sim o próprio LaTeX, que tem uma natural afinidade com a matemática. De facto, a produção de texto em LaTeX é feita declarando funções (designadas "macros") cujo resultado é o texto a imprimir. Veja-se o exemplo do operador unário $cata$, que em LaTeX é a macro unária do mesmo nome declarada no ficheiro auxiliar `cp0809t.sty`.

A inspecção deste ficheiro revelará ao leitor a vantagem de se reproduzir na linguagem do processador de texto a estrutura da linguagem matemática que estamos a usar. Isso revela-se em particular no que diz respeito à apresentação dos cálculos, tal como o que se segue, extraído "verbatim" dos apontamentos teóricos da disciplina [5]:

$$\begin{aligned}
& swap \cdot swap \\
= & \quad \{\ \text{by definition } swap \stackrel{\text{def}}{=} \langle \pi_2, \pi_1 \rangle\ \} \\
& \langle \pi_2, \pi_1 \rangle \cdot swap \\
= & \quad \{\ \text{by } \times\text{-fusion (8)}\ \} \\
& \langle \pi_2 \cdot swap, \pi_1 \cdot swap \rangle \\
= & \quad \{\ \text{definition of } swap \text{ twice}\ \}
\end{aligned}$$

$$
\begin{array}{ll}
 & \langle \pi_2 \cdot \langle \pi_2, \pi_1 \rangle, \pi_1 \cdot \langle \pi_2, \pi_1 \rangle \rangle \\
= & \quad \{ \text{ by } \times\text{-cancellation (6) } \} \\
 & \langle \pi_1, \pi_2 \rangle \\
= & \quad \{ \text{ by } \times\text{-reflexion (7) } \} \\
 & id
\end{array}
$$

A inspecção de filecp0809t.lhs revelará uma estrutura

```
\begin{calculation}
%
     swap \comp swap
%
\just={ .................. }
%
        ......
%
\just={ .................. }
%
        id
%
\end{calculation}
```

cujo 'layout' se aproxima bastante do que é produzido e onde desempenha papel muito relevante a *macro* binária `just` (de *"justificação"*) cujo primeiro argumento é o símbolo a justificar (igualdade de funções, neste caso) e o segundo é o texto justificativo.

5 O que se pretende

O anexo A deste texto contém um enunciado de questões várias sobre a matéria da disciplina, umas de natureza teórica e outras de natureza prática, devendo resultar das primeiras cálculos de programas em estilo *point-free* e das últimas programas em HASKELL que deverão poder ser testados num interpretador de HASKELL. Os alunos devem responder a cada pergunta escrevendo o texto `lhs` *("literate haskell")* que entenderem e como entenderem, ora escrevendo os seus cálculos como acima se explicou, ora embebendo código HASKELL no seu texto. Para imprimir o relatório basta usar a sequência de comandos

```
lhs2TeX -o cp0809t.tex cp0809t.lhs
pdflatex cp0809t
```

(onde se usa `cp0809t.lhs` como exemplo) em que o papel essencial é desempenhado pelo pre-processador `lhs2tex` [3].

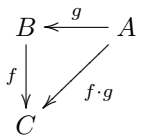
Sugere-se o aproveitamento da fonte LaTeX deste documento (`cp0809t.lhs`) como ponto de partida. O ficheiro fonte do relatório deverá ser enviado à equipa docente seguindo as instruções que serão publicadas na página da disciplina.

Os alunos deverão ainda aproveitar este texto para mostrar que sabem manipular bibliografias (em BibTeX) e índices/glossários (usando `makeindex`). Para se conhecer a vastíssima documentação disponível sobre LaTeX, BibTeX, etc, etc, basta escrever `"latex"`, `"bibtex"`, `"makeindex"` etc num motor de busca e começar a navegar. Ou então visitar o TUG (*TeX Users Group*) em `www.tug.org/`.

[3] Ver `www.informatik.uni-bonn.de/~loeh/lhs2tex`.

6 Desenho de diagramas

Há muitos *pacotes LaTeX* para desenhar diagramas e outros objectos gráficos [1]. Sugere-se, entre estes, o XY-PIC [4], pela sua simplicidade e pelo facto de constar das principais distribuições de LaTeX. Como exemplo, dá-se o diagrama,



extraído de [5]. Bastará inspeccionar o ficheiro `cp0809t.lhs` para ver quão simples é desenhar diagramas como este.

[4] Ver `http://www.tug.org/applications/Xy-pic/`.

A Exercícios a resolver

Como orientação geral para a resolução das questões que se seguem, considerem que as respostas (sejam elas cálculos, provas, código HASKELL, etc) serão tanto mais valorizadas quanto **mais simples** forem. No caso de código HASKELL, procurem usar combinadores que reduzam o número de linhas de código.

O trabalho consta da realização de 10 exercícios, disponibilizados em duas fases para permitir maior flexibilidade na sua resolução.

Seguem-se 10 propostas de exercícios a resolver das quais se devem resolver 8, sendo 5 obrigatórias (marcadas com o sinal "† "). As restantes são à escolha de cada grupo. As valorizaÁıes de questıes ou alíneas obrigatórias, quando existam, s„o naturalmente opcionais.

Exercício 1. Demonstre a lei de fusão do coproduto, cf. (19) no Anexo D, pág. 16, desenhando o diagrama respectivo.

Exercício 2. † Relembre as aulas em que estou a aplicação do algoritmo de Hindley-Milner à inferência de tipos polimórficos em Haskell. Aplique os passos desse algoritmo na dedução do tipo

$$for :: (Integral\ n) \Rightarrow (a \to a) \to a \to n \to a$$

que o GHCi devolve para o combinador for que a biblioteca `Nat.hs` oferece como "fold" associado aos números naturais.

Exercício 3. † Sempre que um gene de um catmorfismo é, como acontece na lei de recursividade múltipla (32), um "split" de duas funções, designa-se esse caso particular de catamorfismo um **mutumorfismo** e usa-se uma notação especial para o identificar, tal como se segue:

$$\lhd h, k \rhd \quad \stackrel{\text{def}}{=} \quad (\!|\langle h, k\rangle|\!) \tag{2}$$

Repare que a lei de de recursividade múltipla (32) pode ser vista como a propriedade universal do combinador $\lhd, \rhd$, lei essa que, aliás, se pode ainda desenvolver mais usando propriedades de produtos (secção D.1).

Pretende-se neste exercício avaliar a sua agilidade na dedução de novas leis do cálculo de programas a partir das já conhecidas. Nesse sentido, derive as leis de cancelamento, fusão e reflexão para mutumorfismos.

Exercício 4. Demonstre a seguinte igualdade onde se apresentam duas formas de calcular a mesma função:

$$for\ (a+)\ 0 = \pi_2 \cdot for\ (\langle \pi_1, \overline{+}\rangle)\ (a, 0) \tag{3}$$

Que função é essa?

Exercício 5. Dado que todos os catamorfismos sobre naturais são da forma $(\!|[\underline{k}, g]|\!)$ mostre que $(\!|[\underline{k}, g]|\!)\, n = g^n k$, onde g^n é n-ésima iteração de g, i.e., $g^0 = id$ e $g^{n+1} = g \cdot g^n$. Note que g é o corpo de um "ciclo-for" que se repete n-vezes, começando com o valor inical k.

Exercício 6. † A função coseno $\cos x : I\!R \to [-1, 1]$ pode ser definida de várias maneiras, entre elas aquela que toma a forma de uma série de Taylor:

$$\cos x \quad = \quad \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n} \quad \forall x \tag{4}$$

A seguinte função, em HASKELL,

$$\begin{aligned}
&coseno :: Double \to Integer \to Double \\
&coseno\ x\ 0 = 1 \\
&coseno\ x\ (n+1) = sig\ (n+1) * (pow2n\ x\ (n+1))\ /\ (fac2n\ (n+1)) \\
&\quad + coseno\ x\ n
\end{aligned}$$

calcula uma aproximação a $cosx$, onde o parâmetro adicional indica o número de parcelas da série de Taylor que estão a ser tomadas em consideração. Por exemplo, enquanto que $coseno\ 1\ 1 = 0.5$, já $coseno\ 1\ 10$ dá o resultado 0.5403023058681397.

Recorde das aulas práticas que podemos "linearizar" a função x^{2n} definindo uma nova função $pow2n\, n = x^{2n}$ e desenvolvendo em

$$\begin{aligned}
&pow2n\ x\ 0 = 1 \\
&pow2n\ x\ (n+1) = x * x * (pow2n\ x\ n)
\end{aligned}$$

Que por sua vez é um catamorfismo de naturais, e a função sig também se pode exprimir através de um catamorfismo no mesmo tipo de dados:

$$sig = cataNat\ [\underline{1}, (-1)*]$$

sendo aqui expressa na sua forma minimal. Já a função que calcula o factorial de $2n$ pode ser também definida por recursão no tipo de dados Nat usando uma outra versão do catamorfismo.

$$\begin{aligned}
&fac2n = \pi_2 \cdot fcata \\
&\quad \textbf{where} \\
&\quad\quad fcata\ 0 = (0, 1) \\
&\quad\quad fcata\ (n+1) = \textbf{let}\ (a, b) = fcata\ n \\
&\quad\quad\quad \textbf{in}\ (a+1, ((2*a)+2) * ((2*a)+1) * b)
\end{aligned}$$

Contudo, a função *coseno* não é muito eficiente. Se recorrer às funções disponíveis na package $System.Time$ para contagem de tempo de CPU verifica que, enquanto o cálculo de $coseno\ 1\ 100$ é quase instantâneo, o de $coseno\ 1\ 1000$ leva cerca de 4 segundos, o de $coseno\ 1\ 2000$ 22 segundos, etc

Apresente em detalhe todos os cálculos que deverá fazer para, recorrendo a leis como a de recursividade múltipla e/ou o seu corolário *banana-split*, obter uma versão mais eficiente de $coseno\ x\ n$.

Exercício 7. † Para resolver este exercício deverá ler primeiro o apêndice B e, em particular, o que se refere ao **triângulo de Sierpinski**:

1. Desenvolva a biblioteca *pointfree* `TLTree.hs` de forma análoga a outras bibliotecas que conhece (*eg.* `BTree.hs`, ...).
2. Exprima as funções `geraSierp` e `apresentaSierp` como anamorfismo e catamorfismo, respectivamente, do tipo `TLTree`.
3. Recorrendo às funções relevantes da biblioteca gráfica, escreva uma função

   ```
   desenha :: [Tri] -> IO()
   ```

   que desenha uma lista de triângulos.
4. Teste o seu programa:

   ```
   desenha (sierpinski (50,300) 256 5)
   ```

Exercício 8. Atente no módulo Google apresentado no apéndice C.

1. Converta a função **groupByKey** para pontwise e apresente a sua definição em Haskell.
2. Recorrendo às leis de (30) consegue comprimir algumas das definções apresentadas no modulo Google?

Exercício 9. Pretende-se implementar um motor de busca para um sistema de ficheiros usando o modelo MapReduce. Suponha para uma dada pesquisa X (uma ou mais palavras) o resultado é a lista dos ficheiros onde essa(s) palavra(s) X ocorre(m) ordenada por número de ocorrências.

Para isso precisamos de construir uma função

$$makeIndex::[(NomeFicheiro, Conteudo)] \to [(Palavra, [(NomeFicheiro, Int)])]$$

que, dados os nomes de ficheiro e o seu conteúdo (texto), pré-compute para cada palavra contida nesses ficheiros a lista dos nomes de ficheiro onde aparece e o número de vezes.

Identifique os parametros m(ap) e r(educe) que calculam esta função.

Exercício 10. † Na secção (5) da biblioteca `LTree.hs` declara-se o tipo $LTree\ a$ como instância da classe $Monad$. Apresente cálculos que provem que as propriedades (34) e (35) são de facto válidas, para este mónade.
Valorização: poderá optar por provar que, em geral, qualquer tipo cuja base é $\mathsf{B}(f,g) = f + \mathsf{F}\,g$ (onde F é um functor arbitrário) forma um mónade, para $\mu = (\!|[id, in \cdot i_2]|\!)$ e $u = in \cdot i_1$. Já agora: que outros mónades estudou que caem nesta classe?

B Gráficos em Haskell

B.1 Fractais

Um *fractal* é uma estrutura matemática que se repete infinitamente, e a que corresponde uma representação visual "infinitamente detalhada". Uma imagem possível é a seguinte: se usarmos dispositivos de ampliação cada vez mais potentes (lupas, microscópios), conseguiremos sempre ver cada vez mais detalhes na figura, que pareciam não existir quando a observávamos com o dispositivo anterior. Os fractais têm aplicações importantes que não serão aqui exploradas, nomeadamente na simulação de fenómenos naturais (por exemplo climatéricos).

Neste trabalho estudar-se-á a geração de fractais por hilomorfismos de determinados tipos indutivos.

B.2 A Biblioteca Gráfica `Graphics.SOE`

Para o desenho dos fractais sugere-se a utilização da biblioteca `Graphics.SOE` [6][5]. Para isso basta adicionar

```
import Graphics.SOE
```

como já foi feito. O desenho de objectos gráficos não é mais do que um caso especial de processamento de *input / output* monádico. Apresentam-se aqui algumas funções básicas suficientes para o presente trabalho.

Antes de mais, vejamos como criar uma janela gráfica utilizando a seguinte função:

```
type Title = String
type Size = (Int,Int)
openWindow :: Title -> Size -> IO Window
```

O seguinte programa abre uma janela com nome "Sierpinski", de dimensão 300 $\times$ 300; escreve nela uma mensagem; espera que uma tecla seja premida; finalmente fecha a janela.

```
main =
  runGraphics (
  do w ← openWindow "Sierpinski" (300,300)
     drawInWindow w (text (100,200) "Aqui está uma janela")
     k ← getKey w
     closeWindow w
  )
```

A função `drawInWindow :: Window -> Graphic -> IO()` desenha um objecto de tipo `Graphic` numa janela. É naturalmente possível desenhar outro tipo de objectos além de texto; o seguinte será útil para o desenho dos fractais:

```
type Point = (Int,Int)
polygon :: [Point] -> Graphic
```

Objectos gerados por esta função correspondem a polígonos descritos por uma lista de pontos.

Para colorir os fractais sugere-se a utilização da seguinte função:

```
withColor :: Color -> Graphic -> Graphic
```

[5] distribuída como parte da *Hugs Graphics Library* em `http://haskell.org/graphics/`

Como exemplo de utilização, apresenta-se uma função que desenha numa janela um triângulo rectângulo isósceles de cor azul, descrito pelas coordenadas de um dos seus vértices e pelo comprimento dos seus catetos:

$$
\begin{aligned}
&\mathit{fillTri} :: \mathit{Window} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int} \to \mathit{IO}\ () \\
&\mathit{fillTri}\ w\ x\ y\ \mathit{size} = \mathit{drawInWindow}\ w\ (\mathit{withColor}\ \mathit{Blue} \\
&\quad (\mathit{polygon}\ [(x, y), (x + \mathit{size}, y), (x, y - \mathit{size}), (x, y)]))
\end{aligned}
$$

Uma nota final: o ponto $(0, 0)$ numa janela gráfica corresponde ao canto *superior esquerdo* da mesma.

B.3 O Triângulo de Sierpinski

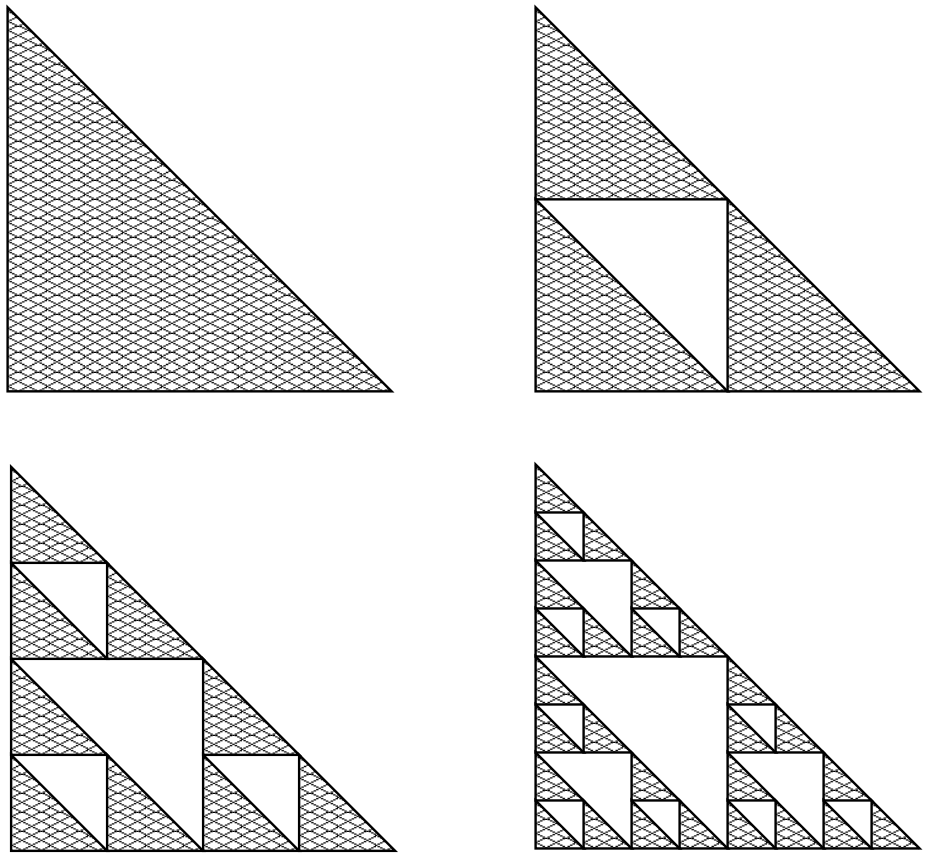


Figura 1: Construção de um triângulo de Sierpinski

O triângulo de Sierpinski é uma figura com carácter fractal que se obtém da seguinte forma: considere-se um triângulo rectângulo e isósceles A cujos catetos têm comprimento s. A estrutura fractal é criada desenhando-se três triângulos no interior de A, todos eles rectângulos e isósceles e com catetos de comprimento $s/2$. Este passo é depois repetido para cada um dos triângulos desenhados, e assim sucessivamente. Os três primeiros passos são apresentados na Fig. 1.

Note-se que um triângulo de Sierpinski é gerado repetindo-se infinitamente o processo acima descrito; no entanto para efeitos de visualização num monitor (de resolução forçosamente finita) será necessário escolher uma representação adequada do triângulo, parando o processo recursivo a um determinado nível.

A figura a desenhar é constituída por um conjunto finito de triângulos todos da mesma dimensão (por exemplo, no quarto triângulo da figura, desenhar-se-iam 27 triângulos). Cada triângulo é geometricamente descrito pelas coordenadas do seu vértice inferior esquerdo e o comprimento dos seus catetos:

$$\begin{aligned}&\textbf{type } \mathit{Side} = \mathit{Int}\\&\textbf{type } \mathit{Tri} = (\mathit{Point}, \mathit{Side})\end{aligned}$$

B.4 Criação de um Triângulo de Sierpinski por um Hilomorfismo

A estrutura recursiva de (uma representação finita de) um triângulo de Sierpinski é captada por uma árvore ternária (cada nó é um triângulo com os respectivos três sub-triângulos). Nas folhas dessa árvore encontram-se os triângulos mais pequenos (todos da mesma dimensão), que deverão ser desenhados. Apenas estes conterão informação de carácter geométrico, tendo os nós da árvore um papel exclusivamente estrutural.

$$\begin{aligned}&\textbf{data } \mathit{TLTree} = \mathit{Leaf}\ \mathit{Point}\ \mathit{Side}\\&\quad \mid \mathit{Node}\ \mathit{TLTree}\ \mathit{TLTree}\ \mathit{TLTree}\end{aligned}$$

A informação geométrica guardada em cada folha consiste nas coordenadas do vértice inferior esquerdo e no lado dos catetos do respectivo triângulo.

A função `sierpinski :: Point -> Side -> Int -> [Tri]` recebe as coordenadas do vértice inferior esquerdo e a dimensão do triângulo exterior, bem como o número de níveis pretendido (critério de paragem do processo de construção do fractal), e constrói uma lista de triângulos a desenhar. Esta função é um hilomorfismo do tipo `TLTree`, composição das seguintes duas funções:

$$\begin{aligned}&\mathit{sierpinski} :: \mathit{Point} \to \mathit{Side} \to \mathit{Int} \to [\mathit{Tri}]\\&\mathit{sierpinski}\ p\ s = \mathit{apresentaSierp} \cdot (\mathit{geraSierp}\ p\ s)\\&\\&\mathit{geraSierp} :: \mathit{Point} \to \mathit{Side} \to \mathit{Int} \to \mathit{TLTree}\\&\mathit{geraSierp}\ (x, y)\ s\ 0 = \mathit{Leaf}\ (x, y)\ s\\&\mathit{geraSierp}\ (x, y)\ s\ n = \mathit{Node}\ (\mathit{geraSierp}\ (x, y)\ (s\ `\mathit{div}`\ 2)\ (n-1))\\&\quad (\mathit{geraSierp}\ (x + s\ `\mathit{div}`\ 2, y)\ (s\ `\mathit{div}`\ 2)\ (n-1))\\&\quad (\mathit{geraSierp}\ (x, y - s\ `\mathit{div}`\ 2)\ (s\ `\mathit{div}`\ 2)\ (n-1))\\&\\&\mathit{apresentaSierp} :: \mathit{TLTree} \to [(\mathit{Point}, \mathit{Side})]\\&\mathit{apresentaSierp}\ (\mathit{Leaf}\ (x, y)\ s) = [((x, y), s)]\\&\mathit{apresentaSierp}\ (\mathit{Node}\ a\ b\ c) =\\&\quad (\mathit{apresentaSierp}\ a) +\!\!+ (\mathit{apresentaSierp}\ b) +\!\!+ (\mathit{apresentaSierp}\ c)\end{aligned}$$

C O modelo MapReduce

A seguir apresenta-se um módulo contendo uma implementação do modelo de programação MapReduce tornado popular pela Google [4]. Resumidamente este modelo baseia-se em aplicar uma função a vários elementos de uma lista de pares chave/valor, agrupar os resultados por chave e finalmente calcular resultados finais para cada para cada elemento resultante da fase de agrupamento.

Note-se que, mesmo estando entre blocos `\begin{code}% ... \end{code}`, esta parte não vai ser considerada pelo interpretador/compilador de Haskell (via `lhs2tex`) por causa do símbolo de percentagem extra. Esta variante permite esconder esta parte do interpretador/compilador de Haskell e ao mesmo tempo apresentá-la em LaTeX. Recorremos a este truque para evitar um ficheiro com duas definições de módulos, algo que não é permitido.

```
%
module Google where
  -- Adatped from Google's MapReduce programming model revisited
  -- by Ralf Laemmel, DOI: http://dx.doi.org/10.1016/j.scico.2007.07.001
import Cp
mapReduce :: (Eq b, Eq a1) =>
  (a2 -> b2 -> [(a1, b)])
  -> (a1 -> [b] -> Maybe b1)
  -> [(a2, b2)]
  -> [(a1, b1)]
mapReduce m r =
  (reducePerKey r)          -- 3. Apply *r* to each group
  . groupByKey              -- 2. Group intermediate data per key
  . (mapPerKey m)           -- 1. Apply *m* to each key/value pair
mapPerKey f =
  concat                    -- 2. Concatenate per-key lists
  . map f^                  -- 1. Map *f* over list of pairs
groupByKey :: (Eq b, Eq a) => [(a, b)] -> [(a, [b])]
groupByKey = foldr ins []
  where ins (a, v) m = consWith (++) a [v] m
reducePerKey r =
  (mapWithKey unJust)       -- 3. Transform type to remove Maybe
  . filterWithKey isJust    -- 2. Remove entries with value Nothing
  . (mapWithKey r)          -- 1. Apply *r* per key
  where
    unJust k x = gJust x    -- Transforms optional into non-optional type
    isJust k (Just v) = True    -- Keep entries of this form
    isJust k Nothing = False    -- Remove entries of this form


%
  -- Example:
main = print
  $ wordOccurrenceCount
  $ insert "doc2" " appreciate the unfold "
  $ insert "doc1" " fold the fold "
  $ []
  -- where
```

```
wordOccurrenceCount = mapReduce m r
  where m = (map (flip (,) 1) . words)   -- each word counts as 1
        r = (Just . sum)   -- compute sum of all counts
        sum = foldl (+) 0


%
  -- Auxiliary functions
gJust (Just v) = v   -- Get contents of Maybe
consWith :: (Eq b, Eq a) => (a -> a -> a) -> b -> a -> [(b, a)] -> [(b, a)]
consWith f k v m
  | lookup k m == Nothing = (k, v) : m
  | otherwise = (k, f v z) : (filter ((/= k) . π1) m)
                where z = gJust (lookup k m)
insert :: (Eq a, Eq b) => b -> a -> [(b, a)] -> [(b, a)]
insert = consWith _
filterWithKey :: (a -> b -> Bool) -> [(a, b)] -> [(a, b)]
filterWithKey p = filter p^
mapWithKey :: (a -> b -> c) -> [(a, b)] -> [(a, c)]
mapWithKey f = map (split π1 f^)
```

D Algumas leis do cálculo funcional

D.1 Produto

$$\textbf{Universal-}\times \qquad k = \langle f, g\rangle \quad \equiv \quad \begin{cases} \pi_1 \cdot k = f \\ \pi_2 \cdot k = g \end{cases} \tag{5}$$

$$\textbf{Cancelamento-}\times \qquad \pi_1 \cdot \langle f, g\rangle = f \quad , \quad \pi_2 \cdot \langle f, g\rangle = g \tag{6}$$

$$\textbf{Reflexão-}\times \qquad \langle \pi_1, \pi_2\rangle \quad = \quad id_{A\times B} \tag{7}$$

$$\textbf{Fusão-}\times \qquad \langle g, h\rangle \cdot f = \langle g \cdot f, h \cdot f\rangle \tag{8}$$

$$\textbf{Absorção-}\times \qquad (i \times j) \cdot \langle g, h\rangle = \langle i \cdot g, j \cdot h\rangle \tag{9}$$

$$\textbf{Def-}\times \qquad f \times g = \langle f \cdot \pi_1, g \cdot \pi_2\rangle \tag{10}$$

$$\textbf{Natural-}\pi_1 \qquad \pi_1 \cdot (f \times g) \quad = \quad f \cdot \pi_1 \tag{11}$$

$$\textbf{Natural-}\pi_2 \qquad \pi_2 \cdot (f \times g) \quad = \quad g \cdot \pi_2 \tag{12}$$

$$\textbf{Functor-}\times \qquad (g \cdot h) \times (i \cdot j) \quad = \quad (g \times i) \cdot (h \times j) \tag{13}$$

$$\textbf{Functor-id-}\times \qquad id_A \times id_B \quad = \quad id_{A\times B} \tag{14}$$

$$\textbf{Eq-}\times \qquad \langle f, g\rangle = \langle h, k\rangle \ \equiv \ f = h \wedge g = k \tag{15}$$

D.2 Coproduto

$$\textbf{Universal-}+ \qquad k = [f, g] \quad \equiv \quad \begin{cases} k \cdot i_1 = f \\ k \cdot i_2 = g \end{cases} \tag{16}$$

$$\textbf{Cancelamento-}+ \qquad [g, h] \cdot i_1 = g \quad , \quad [g, h] \cdot i_2 = h \tag{17}$$

$$\textbf{Reflexão-}+ \qquad [i_1, i_2] \quad = \quad id_{A+B} \tag{18}$$

$$\textbf{Fusão-}+ \qquad f \cdot [g, h] = [f \cdot g, f \cdot h] \tag{19}$$

$$\textbf{Absorção-}+ \qquad [g, h] \cdot (i + j) = [g \cdot i, h \cdot j] \tag{20}$$

$$\textbf{Def-}+ \qquad f + g = [i_1 \cdot f, i_2 \cdot g] \tag{21}$$

$$\textbf{Functor-}+ \qquad (g \cdot h) + (i \cdot j) = (g + i) \cdot (h + j) \tag{22}$$

$$\textbf{Functor-id-}+ \qquad id_A + id_B = id_{A+B} \tag{23}$$

$$\textbf{Eq-}+ \qquad [f, g] = [h, k] \ \equiv \ f = h \ \wedge \ g = k \tag{24}$$

$$\textbf{Lei da troca} \qquad [\langle f, g\rangle, \langle h, k\rangle] \quad = \quad \langle [f, h], [g, k]\rangle \tag{25}$$

D.3 Indução

$$\textbf{Universal-cata} \qquad k = (\!|\beta|\!) \quad \equiv \quad k \cdot in = \beta \cdot (\mathsf{F}\, k) \tag{26}$$

$$\textbf{Cancelamento-cata} \qquad (\!|\alpha|\!) \cdot in = \alpha \cdot \mathsf{F}\, (\!|\alpha|\!) \tag{27}$$

$$\textbf{Reflexão-cata} \qquad (\!|in|\!) = id_{\mu\mathsf{F}} \tag{28}$$

$$\textbf{Fusão-cata} \qquad f \cdot (\!|\alpha|\!) = (\!|\beta|\!) \quad \Leftarrow \quad f \cdot \alpha = \beta \cdot (\mathsf{F}\, f) \tag{29}$$

$$\textbf{Absorção-cata} \qquad (\!|\alpha|\!) \cdot \mathsf{T}\, f \quad = \quad (\!|\alpha \cdot \mathsf{B}(f, id)|\!) \tag{30}$$

$$\textbf{Def-map} \qquad \mathsf{T}\, f \quad = \quad (\!|in_{\mathsf{F}} \cdot \mathsf{B}(f, id)|\!) \tag{31}$$

D.4 Recursividade mútua

$$\textbf{Fokkinga} \qquad \begin{cases} f \cdot in = h \cdot \mathsf{F}\, \langle f, g\rangle \\ g \cdot in = k \cdot \mathsf{F}\, \langle f, g\rangle \end{cases} \quad \equiv \quad \langle f, g\rangle = (\!|\langle h, k\rangle|\!) \tag{32}$$

$$\textbf{“Banana-split”} \qquad \langle (\!|i|\!), (\!|j|\!)\rangle = (\!|(i \times j) \cdot \langle \mathsf{F}\, \pi_1, \mathsf{F}\, \pi_2\rangle|\!) \tag{33}$$

D.5 Mónades

Multiplicação	$\mu \cdot \mu = \mu \cdot \mathsf{F}\,\mu$	(34)
Unidade	$\mu \cdot u = \mu \cdot \mathsf{F}\,u = id$	(35)

Referências

[1] Michel Goossens, Sebastian Rahtz, and Frank Mittelbach. *The LaTeX Graphics Companion*. Addison-Wesley, 1997. ISBN 0-201-85469-4.

[2] D.E. Knuth. *The TEXbook*. Addison-Wesley Publishing Company, 7th edition, 1986.

[3] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[4] Ralf Lämmel. Google's mapreduce programming model — revisited. *Sci. Comput. Program.*, 68(3):208–237, 2007.

[5] J.N. Oliveira. *Program Design by Calculation* . Draft of textbook in preparation (since 2005). Departamento de Informática, Universidade do Minho.

[6] P.Hudak. *The haskell school of expression: Learning functional programming through multimedia*. 2000.

Índice