Trabalho 2: Conversão entre *pointwise* e *point-free* e vice-versa

Métodos de Programação 1

22 de Novembro de 2006

1 Introdução

Como já vimos nas aulas práticas, é possível converter programas do estilo *pointwise* para o estilo *point-free* usando cálculo algébrico. Por exemplo, considere a seguinte definição para a função assocr.

$$\begin{aligned}&\mathsf{assocr} :: ((a, b), c) \to (a, (b, c))\\ &\mathsf{assocr}\ ((x, y), z) = (x, (y, z))\end{aligned}$$

Recorrendo às leis apresentadas nas aulas, e que podem ser consultadas na respectiva sebenta, é possível converter esta definição *pointwise* para o estilo *point-free* da seguinte forma.

$$\begin{aligned}&\quad \mathsf{assocr}\ ((x, y), z) = (x, (y, z))\\ \Leftrightarrow &\quad \{\ \textsc{Elim}\text{-}\times\ \}\\ &\quad \mathsf{assocr}\ (w, z) = (\mathsf{fst}\ w, (\mathsf{snd}\ w, z))\\ \Leftrightarrow &\quad \{\ \textsc{Elim}\text{-}\times\ \}\\ &\quad \mathsf{assocr}\ x = (\mathsf{fst}\ (\mathsf{fst}\ x), (\mathsf{snd}\ (\mathsf{fst}\ x), \mathsf{snd}\ x))\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-}\circ\ \}\\ &\quad \mathsf{assocr}\ x = ((\mathsf{fst} \circ \mathsf{fst})\ x, ((\mathsf{snd} \circ \mathsf{fst})\ x, \mathsf{snd}\ x))\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-}\triangle\ \}\\ &\quad \mathsf{assocr}\ x = ((\mathsf{fst} \circ \mathsf{fst})\ x, (\mathsf{snd} \circ \mathsf{fst} \triangle \mathsf{snd})\ x)\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-}\triangle\ \}\\ &\quad \mathsf{assocr}\ x = (\mathsf{fst} \circ \mathsf{fst} \triangle (\mathsf{snd} \circ \mathsf{fst} \triangle \mathsf{snd}))\ x\\ \Leftrightarrow &\quad \{\ \textsc{Ext}\text{-}=\ \}\\ &\quad \mathsf{assocr} = \mathsf{fst} \circ \mathsf{fst} \triangle (\mathsf{snd} \circ \mathsf{fst} \triangle \mathsf{snd})\\ \Leftrightarrow &\quad \{\ \textsc{Nat}\text{-}\mathsf{id}\ \}\\ &\quad \mathsf{assocr} = \mathsf{fst} \circ \mathsf{fst} \triangle (\mathsf{snd} \circ \mathsf{fst} \triangle \mathsf{id} \circ \mathsf{snd})\\ \Leftrightarrow &\quad \{\ \textsc{Def}\text{-}\times\ \}\\ &\quad \mathsf{assocr} = \mathsf{fst} \circ \mathsf{fst} \triangle (\mathsf{snd} \times \mathsf{id})\end{aligned}$$

Nesta derivação usamos uma notação alternativa para o combinador *split*: $f \triangle g$ em vez de $\langle f, g\rangle$. A conversão da definição *point-free* para o estilo *pointwise* obtém-se invertendo esta demonstração. Para grande parte das definições é possível automatizar estas derivações, pois consistem essencialmente na aplicação sistemática das definições dos combinadores.

2 Representação das funções

Durante o processo de derivação uma função será definida através de uma conjunção de equações.

```
data Def = Exp :=: Exp | Def :∧: Def
```

Nesta declaração foram usados construtores infixos para facilitar a leitura. As expressões referidas numa equação podem estar definidas no estilo *pointwise* (usando variáveis e aplicação de funções), no estilo *point-free* (recorrendo ao conjunto usual de combinadores e funções primitivas), ou numa mistura dos dois em passos intermédios. Sendo assim, vamos usar o seguinte tipo de dados para representar as expressões:

```
data Exp = Id | Exp :∘: Exp
   | Bang | Const Exp
   | Fst | Snd | Exp :△: Exp | Exp :×: Exp
   | Inl | Inr | Exp :▽: Exp | Exp :+: Exp
   | Cond Exp Exp Exp
   | Fun String | Num Int | ...
   | Var String | Pair Exp Exp | Exp :@: Exp
```

Usando este tipo de dados a função assocr é representada no estilo *pointwise* da seguinte forma:

```
assocr :: Def
assocr = ((Fun "assocr") :@: (Pair (Pair (Var "x") (Var "y")) (Var "z")))
           :=:
         (Pair (Var "x") (Pair (Var "y") (Var "z")))
```

O construtor *Fun* é usado para referir funções constantes: quando aparece no lado esquerdo de uma equação serve para referir a própria função a ser definida. De igual forma, o construtor *Num* é usado para referir valores constantes do tipo inteiro, podendo existir construtores semelhantes para constantes de outros tipos.

Considera-se que uma definição está no estilo *point-free* quando não usa os três últimos construtores apresentados: *Var* para representar variáveis, *Pair* para construir pares explicitamente e :@: para representar a aplicação de uma função a um argumento. Por exemplo, de acordo com a derivação acima, a função assocr pode ser representada no estilo *point-free* da seguinte forma:

```
assocr :: Def
assocr = (Fun "assocr") :=: ((Fst :∘: Fst) :△: (Snd :×: Id))
```

As funções que envolvem somas quando convertidas para o estilo *pointwise* dão normalmente origem a várias equações. Por exemplo a função undistr pode ser representada em *point-free* como

```
undistr :: Def
undistr = (Fun "undistr") :=: ((Id :×: Inl) :▽: (Id :×: Inr))
```

e em *point-wise* como

```
undistr :: Def
undistr = (((Fun "undistr") :@: (Inl :@: (Pair (Var "x") (Var "y"))))
          :=:
          (Pair (Var "x") (Inl :@: (Var "y"))))
          :∧:
          (((Fun "undistr") :@: (Inr :@: (Pair (Var "x") (Var "y"))))
          :=:
          (Pair (Var "x") (Inr :@: (Var "y"))))
```

3 Tarefas a realizar

Neste trabalho deverão ser implementadas no mínimo as seguintes funções:

- *pfpw* :: *Def* $\rightarrow$ *IO Def* que converte uma definição do estilo *point-free* para o estilo *pointwise*. O resultado é do tipo *IO Def* porque se pretende que essa função apresente todos os resultados intermédios e as leis usadas na derivação.
- *pwpf* :: *Def* $\rightarrow$ *IO Def* que converte uma definição do estilo *pointwise* para o estilo *point-free*. O resultado é do tipo *IO Def* pela mesma razão da função *pfpw*.

Estas funções deverão funcionar correctamente para a maior parte das funções não-recursivas apresentadas nas aulas práticas, como por exemplo as funções assocr e undistr.

Para obter nota superior a Bom devem implementar algumas funcionalidades extra, como por exemplo:

- Extender o tipo das expressões por forma a permitir funções recursivas sobre listas. Nomeadamente, deverão ser incluidos os construtores *In*, *Out*, *Nil*, *Cons* e *Cata*. As funções *pfpw* e *pwpf* devem ser modificadas por forma a converter catamorfismos para funções explicitamente recursivas e vice-versa. Naturalmente, os mais ambiciosos poderão tentar converter funções recursivas definidas sobre qualquer tipo declarado pelo utilizador.
- Escrever funções de *parsing* e *pretty-printing* de definições Haskell para o tipo *Def* e vice-versa. Para tal deverão usar o *package Language.Haskell* do GHC, que já disponibiliza essas funções para um tipo que representa a sintaxe abstracta desta linguagem. Notem que uma definição vai corresponder a um elemento do tipo *HsDecl* construído com *HsFunBind*. Equipados com estas funções podem desenvolver uma ferramenta que, dado um ficheiro Haskell, tenta converter todas as definições nele presentes para o estilo *point-free* e vice-versa.

O trabalho deve ser entregue no *site* respectivo até ao dia 17 de Dezembro. Deve ser realizado em grupos de 3 alunos e o relatório escrito em *literate* Haskell. Este documento foi gerado a partir de uma fonte *literate* Haskell usando a ferramenta `lhs2tex`:

`http://www.informatik.uni-bonn.de/~loeh/lhs2tex/`

Boa Sorte!