Trabalho Prático nº1

Métodos de Programação I

2003/2004

Resumo

O objectivo deste trabalho é realizar um pré-processador para programas na linguagem *Haskell* por forma a que sejam gerados de forma automática as funções `cata`, `ana` e `hylo` para tipos de dados arbitrários definidos pelo utilizador.

1 Preâmbulo

Este trabalho deve ser realizado por grupos com um máximo de três alunos, e deve ser submetido de acordo com as instruções divulgadas na página da disciplina antes da data limite aí mencionada.

2 Introdução

```
module List where

import Mpi
```

Considere-se a declaração de tipo `List` (cfr. `RList` da bblioteca `mpi.hs`)

```
data List a = Nil () | Cons (a,List a)
```

O significado deste tipo de dados determina que exista um isomorfismo entre os conjuntos:

$$\text{List a} \cong 1 + a \times (\text{List a})$$

sendo o par de funções que testemunha esse isomorfismo

$$\begin{aligned} \text{inList} &: 1 + a \times (\text{List a}) \to \text{List a} \\ \text{outList} &: \text{List a} \to 1 + a \times (\text{List a}) \end{aligned}$$

Também associados a este tipo de dados é possível identificar "padrões de recursividade" que nos permitem definir funções que manipulem valores desse tipo. Tal como o `foldr` captura a essência de um padrão de recursividade sobre as listas primitivas do *Haskell*, podemos considerar o que designamos por *catamorfismos* que percorrem toda a estrutura do tipo de dados definido. Em diagrama:

$$\begin{array}{ccc} \text{List a} & \xrightarrow{\text{outList}} & 1 + a \times \text{List a} \\ {\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|g|\!)} \\ X & \xleftarrow{\quad g \quad} & 1 + a \times X \end{array}$$

De forma dual definem-se os *anamorfismos* e, como a composição de um catamorfismo com um anamorfismo, os *hilomorfismos*.

Em resumo, da declaração de um tipo dados (como `List` apresentado atrás) podemos derivar um conjunto de combinadores que nos permitirão definir funções que manipulem valores desse tipo, nomeadamente:

```
inList :: Either () (a,List a) -> List a
--inList = either Nil Cons
inList (Left ()) = Nil ()
inList (Right (x,l)) = Cons (x,l)

outList :: List a -> Either () (a,List a)
outList (Nil ()) = Left ()
outList (Cons (x,l)) = Right (x,l)

recList :: (b->c) -> Either () (a,b) -> Either () (a,c)
recList f = id -|- id >< f

cataList :: (Either () (a,b) -> b) -> List a -> b
cataList f = f . recList (cataList f) . outList

anaList :: (b -> Either () (a,b)) -> b -> List a
anaList f = inList . recList (anaList f) . f

hyloList :: (Either () (a,b) -> b) ->
            (c -> Either () (a,c)) -> c -> b
hyloList f g = cataList f . anaList g
```

Uma vez definidos estes combinadores, estamos habilitados a definir funções como as que tem vindo a ser estudadas neste curso. Por exemplo:

```
fact = hyloList f g
  where f = either (const 1) (uncurry (*))
        g 0 = Left ()
        g n = Right (n,n-1)
```

3 Trabalho a Realizar

Para um tipo de dados `T` lido de um ficheiro, pretende-se gerar automaticamente as funções `inT`, `outT`, `recT`, `cataT`, `anaT` e `hyloT`.

Para a tarefa de ler um ficheiro com código *Haskell* e extrair daí a informação das declarações de tipos é fornecido um ficheiro que utiliza bibliotecas disponibilizadas pelo GHC (ver adiante). Assim, a funcionalidade que deverá ser implementada neste trabalho consiste em gerar as funções referidas a partir da informação armazenada em estruturas de dados adequadas.

As declarações de tipo deverão ser representadas utilizando os seguintes tipos *Haskell*:

```
-- Tipo de dados para conter a informação de uma declaração de tipo.
data TDecl = MkTD  tDesc :: String,      -- Nome do tipo
                   tArgs :: [String],    -- Argumen-
tos (variáveis de tipo)
                   tConsts :: [CDecl]    -- Construtores
                  deriving Eq

-- Tipo de dados para conter a informação de um construtor.
data CDecl = MkCD  cDesc :: String,      -- Nome do construtor
                   cArgs :: [Type]       -- Tipo dos argumentos
                  deriving (Eq,Show)

-- Tipo de dados para armazenar um tipo Haskell
data Type = TUnit                   -- Tipo ()
          | TVar String             -- Variável de tipo
          | TList Type              -- Listas
          | TTuple [Type]           -- Tuplos
          | TArrow Type Type        -- Espaço de funções (não considerar...)
          | TFApp String [Type]     -- Aplicação de um constru-
tor de tipo
            deriving (Eq,Show)
```

A título de exemplo, a declaração de tipo `List` é representada pelo valor:

```
tList = MkTD  tDesc = "List",
              tArgs = ["a"],
              tConsts = [cNil,cCons]

  where cNil = MkCD  cDesc = "Nil",
                     cArgs = [TUnit]

        cCons = MkCD  cDesc = "Cons",
                      cArgs = [(TTuple [TVar "a",TFApp "List" [(TVar "a")]])]
```

Numa primeira fase, é conveniente considerar algumas simplificações sobre a forma dos tipos que irão ser processados:

- Todos os construtores dispõem de um e um só tipo de dados como argumento.
- Os tipos de dados não fazem referência a quaisquer outros construtores de tipos (e.g. não fazem uso de listas, ou de outros tipos definidos pelo utilizador).

4 Material fornecido como «kit» para o projecto

Entre na página da disciplina e descarregue, descomprimindo-o (e.g. via `unzip`), o ficheiro `mpi0304mp.zip` que contém o respectivo material pedagógico. Para além de outros ficheiros relevantes para a disciplina, deverá obter:

1. `mpi0304t1.lhs` - trata-se do ficheiro que está a ler neste momento, escrito em «*literate Haskell*». Isto significa que:

- se o carregar no *Hugs* ou *GHCi*, o interpretador carregará o código *Haskell* nele contido e interpretá-lo-á.
- se o processar via LaTeX (ou PDFLaTeX) obterá este mesmo documento em *PDF*. Sugestão: experimente

```
latex mpi0304t1.lhs
dvips -o mpi0304t1.ps mpi0304t1
ps2pdf mpi0304t1.ps
```

  ou, mais simplesmente, `pdflatex mpi0304t1.lhs` [1].

2. Com vista a completar a funcionalidade do projecto fornece-se código *Haskell* para extrair de um módulo as declarações de tipo aí contidas. Mais precisamente, são disponibilizados os ficheiros:

   `HaPar.hs`: Faz uso do parser do *Haskell* (incluido na versão 6.01 do GHC) para extrair as declarações de tipo de um ficheiro contendo um módulo Haskell. Requer a opção `-package haskell-src`.

   `TDecl.hs`: Define os tipos de dados referidos atrás (`TDecl`, `CDecl` e `Type`) juntamente com algumas funções para visualização de valores desses tipos (instâncias da classe `Show`).

   O primeiro destes ficheiros requer a utilização do GHC. No entanto, a sua utilização não interfere com o desenvolvimento da funcionalidade pedida.

5 Valorizações

Alguns pontos sugeridos para melhorar o trabalho.

5.1 Admitir construtores sem argumentos

Em *Haskell* são admitidos construtores sem argumentos. Neste trabalho, essa possibilidade é facilmente incorporada ao identificarmos um construtor sem argumentos com um construtor com argumento de tipo `()` na construção das funções `in` e `out` (ver exemplo no ponto seguinte).

5.2 Admitir construtores *curried*

Outra prática comum em *Haskell* é declarar os construtores com múltiplos argumentos na sua versão *curried*. De facto, a declaração do tipo de listas mais natural para um programador *Haskell* seria:

```
data List' a = Nil' | Cons' a (List' a)
```

Mais uma vez, facilmente se ajustam as funções `in` e `out` por forma a acomodar esta diferença. Ilustrando para o caso do `outList'` teríamos:

```
outList' :: List' a -> Either () (a,List' a)
outList' Nil' = Left ()
outList' (Cons' x l) = Right (x,l)
```

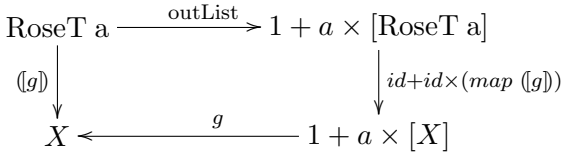
[1] Se não está habituado a LaTeX: em *Linux*, faz parte da distribuição standard e é só experimentar; em *Windows*, sugere-se a instalação de *MiKTeX* (`http://www.miktex.org/`).

5.3 Suportar a utilização de listas nos argumentos de construtores

Ocasionalmente, temos necessidade de considerar tipos mais complexos nos argumentos dos construtores, como as listas primitivas do *Haskell.* Um exemplo é o tipo *Rose Trees* já apresentado nas aulas teórico-práticas.

```
data RoseT a = Vazia | Node a [RoseT a]
```

A utilização das listas manifesta-se na utilização do `map`. Ilustrando no diagrama do catamorfismo, temos:



Permitir a utilização deste tipo de declarações passa então por ajustar convenientemente a definição da função `recT`.

5.4 Suportar tipos de dados dependentes de outros tipos

Uma generalização do problema referido no ponto anterior consiste em considerar tipos de dados que dependam de um outro tipo de dados fornecido pelo utilizador (e.g. definir as *Rose Trees* utilizando o tipo `List` apresentado atrás). A estratégia seguida no caso das listas primitivas pode ser generalizada para cobrir também este caso.