Linguagens Declarativas
Segunda Lista de Exercícios

Rodrigo Bonifácio

2 de abril de 2015

Introdução

Nessa segunda lista de exercício você deve resolver os problemas de acordo com as especificações das funções descritas neste documento. A entrega das soluções deve ser feita até o dia 19/04/2015. O primeiro problema vale 20% da nota deste trabalho; enquanto que o segundo problema vale 80% da nota.

Problema 01: Validação de Cartões de Crédito

Fonte: Noam Zilberstein. *CIS 194: Introduction to Haskell*. University of Pennsylvania. 2015.

A maior parte das operadoras de cartões de crédito utilizam uma fórmula de checksum conhecida como Luhn Algorithm, para distinguir números válidos de cartão de crédito de uma coleção aleatória de dígitos (ou erros de digitação em uma entrada de um número de cartão de crédito).

O algoritmo consiste nos seguintes passos:

(1) Dado um número de cartão de crédito, multiplique por dois todos os dígitos em uma posição par, no sentido direita para a esquerda e considerando o primeiro índice como sendo 1. Ou seja, o último dígito de um número não é alterado; o segundo último dígito é multiplicado por 2; o terceiro último dígito não é alterado; e assim por diante. Por exemplo, considerando o número `[5, 5, 9, 4]`, o resultado desse passo é `[10, 5, 18, 4]`.

(2) Compute o somatório dos **dígitos** da lista de números resultante no item (1). Ou seja, esse somatório *estilizado* de `[10, 5, 18, 4]` deve levar ao valor `(1 + 0) + 5 + (1 + 8) + 4 = 19`.

(3) Calcule o resto da divisão do somatório obtido no item (ii) pelo valor 10. No exemplo acima, o resto da divisão é o valor 9. Quando o resultado do resto da divisão é 0, o número do cartão de crédito é válido.

Implemente o algoritmo de Luhn considerando a composição das seguintes funções:

```
luhn :: Integer → Bool
luhn card = ⊥
  -- exemplo: luhn 5594589764218858 = True
  -- exemplo: luhn 1234567898765432 = False
sumDigits :: [Integer] → Integer
sumDigits xs = ⊥
  -- exemplo: sumDigits [10, 5, 18, 4] = 19
doubleEveryOther :: [Integer] → [Integer]
doubleEveryOther xs = ⊥
  -- exemplo: doubleEveryOther [4, 9, 5, 5] = [4, 18, 5, 10]
  -- exemplo: doubleEveryOther [0, 0] = [0, 0]
```

Mini linguagem de programação funcional

A estrutura de uma linguagem de programação funcional pode ser diretamente mapeada em um tipo algébrico em Haskell que representa expressões, como o *datatype* `Exp` ilustrado logo abaixo. Note que esta *mini-linguagem* de programação funcional suporta operações sobre valores inteiros e booleanos, referências a expressões nomeadas (`RefId Id`), expressões do tipo `Let` e aplicação de funções (`App Id Args`).

```
data Exp = IConst Int
   | BConst Bool
   | And Exp Exp
   | Or Exp Exp
   | Not Exp
   | Add Exp Exp
   | Sub Exp Exp
   | Mult Exp Exp
   | Div Exp Exp
   | Let Id Exp Exp
   | RefId Id
   | App Id Args
deriving (Show)
```

Como a linguagem suporta a aplicação de funções, precisamos (a) ter uma representação para declarar funções (`FuncDecl`) e (b) passar uma lista de declarações de funções para as funções que verificam os tipos (`baseType`) e avaliam (`eval`) expressões. Os tipos de dados usados estão definidos a seguir.

```
type Id = String
data Type = IntType
```

```
    | BooleanType
    | Undefined
  deriving (Show, Eq)
data Value = IntValue Int
    | BooleanValue Bool
  deriving (Show, Eq)
type FormalArgs = [(Id, Type)]
type Args = [Exp]
type Binding = (Id, Exp)
type Env = [Binding]
data FuncDecl = FuncDecl Id FormalArgs
baseType :: Exp → Env → [FuncDecl] → Type
baseType exp env decls = ⊥
typeCheck :: Exp → Env → [FuncDecl] → Bool
typeCheck exp env decls = ⊥
eval :: Exp → Env → [FuncDecl] → Value
eval exp env decls = ⊥
```

Atividades: implemente as seguintes características

- Para cada um dos tipos de expressão, implemente a semântica da linguagem com a função de interpretação `eval` e a verificação de tipos `typeCheck` com o auxílio da função `baseType`.
- Considere a função definida, na nossa linguagem, como:

```
f :: FuncDecl
f = FuncDecl "f" ["p"] (RefId "n")
```

  Note que a função `f` possui um identificador `n` livre. Considere a seguinte expressão:

```
  -- exp = let x = 5 in f 3
exp :: Exp
exp = Let "n" (IConst 5) (App "f" [(IConst 3)])
```

  A implementação do interpretador pode utilizar a semântica de **escopo dinâmico**; e com isso, a avaliação da expressão `exp` reduziria para o valor `IntValue 5`. Outra alternativa segue a semântica de **escopo estático**, onde o escopo de um identificador corresponde a uma região sintaticamente delimitada (e fora do contexto de execução); e um erro *ideniti-ficador não declarado* deveria ser reportado na avaliação da expressão `exp`. Certifique-se que a sua implementação suporta a semântica de escopo estático.

- Implemente uma expressão `If-Then-Else` e certifique-se que a implementação proposta para aplicação de funções suporta a definição de funções recursivas.

Sugestão: Resolva o primeiro item de forma iterativa. Comece com a construção do interpretador pelas expressões mais simples, até concluir a implementação das expressões mais complexas. Podemos usar o forum da disciplina no ambiente Moodle para discutirmos o desenho da solução e sugiro, caso necessário, vocês usarem a primeira edição do livro "Programming Languages: Application and Interpretation" (Shriram Krishnamurthi) como referência.