Implementação de Recurção e avaliação *Lazy* em Haskell

Luisa Sinzker Fantin, 14/0151893
João Pedro Silva Sousa, 15/0038381
Rafael Oliveira de Souza, 15/0081537

28 de junho de 2017

1 Introdução

Dentro do universo das linguagens de programação é possível dividi-las em várias classificações diferentes, seja de acordo com o paradigma de programação utilizado por essa linguagem (como funcional e procedural), com relação aos tipos (forte ou fracamente tipada, estática ou dinamicamente tipada), às estratégias de avaliação utilizada na linguagem (*eager* ou *lazy*) e diversas outras formas.

1.1 Introdução teórica

Dos mecanismos (ordem) de avaliação de uma linguagem de programação, a estratégia *lazy* é bastante conhecida e utilizada na linguagem Haskell. A avaliação *lazy*, na ausência de efeitos colaterais, possui semântica muito parecida com uma avaliação na ordem convencional.

Basicamente, a estratégia *lazy* é uma fusão entre métodos de avaliação não rigorosos (*non-strict*) com um mecanismo que evita a avaliaçã repetida de expressões as quais já se conhece o resultado (chamado de *sharing*).

Uma linguagem é dita *strict* se todas as funções são sempre estritas, ou seja, uma função só será definida se todos os seus argumentos são conhecidos (avaliados). Uma linguagem não-estrita não possui esse requisito, ou seja, podem existir funções definidas mesmo que nem todos os seus argumentos são conhecidos [3].

O mecanismo *sharing* baseia-se na construção de uma tabela, que mapeia cada argumento com a respectiva expressã já avaliada.

1.2 Contextualização

O foco deste trabalho consiste na implementação de um interpretador que possua estratégia *lazy* de avaliação, além de suporte a chamadas de funções recursivas.

2 Visão geral da linguagem

A linguagem LFCFDLazy fornecida possuia suporte a expressões identificadas (LET), referências à identificadores e funções de alta ordem e suas avaliações (mecanismo de expressões lambda).

Para que seja possível alcançãr o objetivo, foram adcionados ao interpretador da linguagem suporte à avaliações de condicionais (necessárias para a implementação da recursão) e a própria recursão [2].

3 Estudo do interpretador

```
module LFCFDLazy where
  -- import Test.Unit
```

3.1 Defnição dos tipos e estruturas

```
type Id = String
type Env = [(Id, ValorE)]
type DefredSub = [(Id, Expressao)]
data ValorE = VInt Int
    | FClosure Id Expressao Env
    | EClosure Expressao Env
   deriving (Show, Eq)
data Expressao = Valor Int
```

```
  | Soma Expressao Expressao
  | Subtracao Expressao Expressao
  | Multiplicacao Expressao Expressao
  | Divisao Expressao Expressao
  | Let Id Expressao Expressao
  | Ref Id
  | Lambda Id Expressao
  | Aplicacao Expressao Expressao
  | If0 Expressao Expressao Expressao
  | Rec Id Expressao Expressao
 deriving (Show, Eq)
```

O tipo `Id` é apenas uma renomeação para um identificador, que pode ser de uma variável ou de uma função recursiva. O tipo `Env` é o ambiente de mapeamento (tupla) entre os identificadores e suas respectivas expressões associadas. O tipo `DefredSub` é o ambiente de substituições postergadas que, analogamente mapeiam identificadores à expressões porém, com a diferença que elas ainda não foram avaliadas.

3.2 Funções de pesquisa

```
pesquisar :: Id → Env → ValorE
pesquisar v [] = error "Variavel nao declarada."
pesquisar v ((i, e) : xs)
    | v ≡ i = e
    | otherwise = pesquisar v xs
searchApp :: Id → ValorE → Env → Env
searchApp n v [] = [(n, v)]
searchApp n v ((i, e) : xs)
    | n ≡ i = []
    | otherwise = searchApp n v xs
```

As funções *pesquisar :: Id → Env → ValorE* e *searchApp :: Id → ValorE → Env → Env* servem para procurar expressões mapeadas nos ambientes de substituição de identificadores e funções recursivas, respectivamente.

3.3 Funções auxiliares

```
avaliacaoStrict :: ValorE → ValorE
avaliacaoStrict (EClosure e env) = avaliacaoStrict (avaliar e env)
avaliacaoStrict e = e

avaliarExpBin :: Expressao → Expressao → (Int → Int → Int) → Env → ValorE
avaliarExpBin e d op env = VInt (op ve vd)
  where
    (VInt ve) = avaliacaoStrict (avaliar e env)
    (VInt vd) = avaliacaoStrict (avaliar d env')
    env' = case e of
      (Ref v) → ((v, VInt ve) : env)
      otherwise → env
```

A função *avaliacaoStrict :: ValorE → ValorE* realiza uma avaliação de um `EClosure` (closure de uma expressão). Caso o *closure* a ser avaliado por essa função seja um *closure* de uma função ou de um valor inteiro, a funçã simplesmente retorna a própria expressão.

A função *avaliarExpBin :: Expressao → Expressao → (Int → Int → Int) → Env → ValorE* é utilizada para realizar a avaliação *lazy* das expressoes aritméticas binárias (adição, subtração, multiplicação e divisão). A função recebe como argumentos as duas expressões que se deseja calcular, o operador e o ambiente de mapeamento de identificadores e expressões já avaliadas.

O método de avaliação *sharing* é implementado na sentençã `case` dessa função: caso a expressão disposta do lado direito da operação seja igual à expressão do lado esquerdo, ela não é avaliada uma segunda vez, é realizada a recuperação da avaliação da expressão do lado direto no ambiente [1].

3.4 Interpretador

```
avaliar :: Expressao → Env → ValorE
avaliar (Valor n) _ = VInt n
avaliar (Soma e d)          env = avaliarExpBin e d (+) env
avaliar (Subtracao e d)     env = avaliarExpBin e d (−) env
avaliar (Multiplicacao e d) env = avaliarExpBin e d (∗) env
avaliar (Divisao e d)       env = avaliarExpBin e d div env
avaliar (Let v e c)         env = avaliar (Aplicacao (Lambda v c) e) env
```

```
avaliar (Ref v)            env = avaliacaoStrict (pesquisar v env)
avaliar (Lambda a c)       env = FClosure a c env
avaliar (Aplicacao e1 e2) env =
  let
    v = avaliacaoStrict (avaliar e1 env)
    e = EClosure e2 env
  in case v of
    (FClosure a c env') -> avaliar c ((a, e) : env')
    otherwise -> error "Tentando aplicar uma expressao" ++
      "que nao eh uma funcao anonima"
avaliar (If0 v e d)        env
  | avaliar v env == VInt 0 = avaliar e env
  | otherwise = avaliar d env
avaliar (Rec nome e1 e2) env =
  let
    v = avaliacaoStrict (avaliar e1 env)
    e = EClosure e2 env
    env2 = (searchApp nome v env) ++ env
  in case v of
    (FClosure a c env') -> avaliar c ((a, e) : env2)
    otherwise -> error "Tentando aplicar uma expressao" ++
      "que nao eh uma funcao anonima"
```

Esse é o interpretador implementado, já com as adições requeridas e necessárias para o correto funcionamento da estratégia de avaliação *lazy* e suporte a chamadas de funções recursivas.

As principais modificações relativas à estratégia de avaliação *lazy* foram feitas na função `avaliarExpBin`, já que a maioria das expressões são reduzidas às operações básicas da aritmética, conforme detalhado na seção 3.3.

4 Testes

4.1 Recursão

Exemplo 1: com ambiente vazio

Pode-se definir uma função fatorial dessa maneira:

```
-- definicao da funcao fatorial
```

```
base = Valor 1
recursao = Subtracao (Ref "x") (Valor 1)

fac_fail = (Multiplicacao (Ref "x") (Rec "fac" (Ref "fac") recursao))

fac_if = If0 (Ref "x") base fac_fail

fac_fun = Lambda "x" fac_if

def_fac = Rec "fac" fac_fun
```

A recursão é utilizada para calcular o fatorial de um valor. Como definido na construção do tipo *Expressao*, uma expressãoo recurisva possui um nome (identificador) e duas expressões, chamadas e_1 e e_2. A expressão e_1 define a função recursiva e a expressão e_2 é o argumento destino à função.

```
e1 = def_fac
e2 = Valor 4

avaliar (def_fac e2) [ ]
```

Primeiramente, e_1 é avaliado e a expressão `Lambda "x"(If0 (Ref "x") (Valor 1) (Multiplicacao (Ref "x") (Rec "fac"(Ref "fac") (Subtracao (Ref "x") (Valor 1))))))` é transformada em um `ValorE`.

A avaliação de e_1 em um ambiente vazio retorna o *closure* `FClosure "x"(If0 (Ref "x") (Valor 1) (Multiplicacao (Ref "x") (Rec "fac"(Ref "fac") (Subtracao (Ref "x") (Valor 1))))) [ ]`, enquanto e_2 é transformado em um *closure* `EClosure e2 [ ]`.

O ambiente é enriquecido com o nome e o corpo da função utilizados na avaliação de `Rec "fac"`: `["fac", (FClosure "x"(If0 (Ref "x") (Valor 1) (Multiplicacao (Ref "x") (Rec "fac"(Ref "fac") (Subtracao (Ref "x") (Valor 1))))))]`.

Caso a avaliação de e_1 seja um `FClosure`, o ambiente é enriquecido com o identificador e seu respectivo `ValorE`.

Uma função recursiva, por possui normalmente um ou mais casos base, seguidos da aplicação da recursão, deve conter uma condição de parada para que não ocorra recursões infinitas; essa condição de parada é determinada pela expressão `If0`.

A avaliação da expressão `Rec` é executada até que a expressão `Subtracao (Ref "x") (Valor 1)` seja equivalente ao `Valor 0`, ou `VInt 0` (condição de parada).

Os valores obtidos a cada chamada da recusão são computados e conclui-se que, para o caso de teste apresentado, o resultado será um `VInt 24` ($4 \times 3 \times 2 \times 1$).

Exemplo 2: com ambiente enriquecido

Supondo o ambiente populado `amb = [("y", VInt 5)]`:

```
amb = [("y", VInt 5)]

e2 = Ref "y"

avaliar (def_fac e2) amb
```

Analogamente, om o ambiente enriquecido, a avaliação do fatorial de `Ref "y"` é calculado utilizando o ambiente declarado.

O *closure* resultante da avaliação de e_1 permanece o mesmo, com a diferença na avaliação de e_2: `EClosure (Ref "y") [("y", VInt 5)]`.

Ao final, o resultado retornado é `VInt 120` ($5 \times 4 \times 3 \times 2 \times 1$).

4.2 Avaliação *lazy*

Para demonstrar a avaliação *lazy*, a expressão `avaliar Let "x"(Soma(Valor 3)(Valor 4)) ((Soma(Ref"x")(Ref"x"))) []` foi avaliada seguindo o interpretador aqui apresentado. A alteração no interpretador original que evita a reavaliação de uma expressão já previamente avaliada é apresentada a seguir:

```
avaliarExpBin :: Expressao -> Expressao -> (Int -> Int -> Int) -> Env -> ValorE
    avaliarExpBin e d op env = VInt (op ve vd)
        where
            (VInt ve) = avaliacaoStrict (avaliar e env)
            (VInt vd) = avaliacaoStrict (avaliar d env')
            env' = case e of
                (Ref v) -> ((v, VInt ve):env)
                otherwise -> env
```

Após uma variável ser computada, seu valor é armazenado no "ambiente", dessa forma, caso ela apareça novamente, o interpretador já terá o resultado final avaliado. A demonstração está mostrada a seguir:

```
avaliar Let "x" (Soma(Valor 3)(Valor 4)) ((Soma(Ref"x")(Ref"x"))) []
avaliar (Aplicacao (Lambda "x" (Soma(Ref "x")(Ref "x"))) (Soma(Valor 3)(Valor 4))
   v = avaliacaoStrict (avaliar (Lambda "x" (Soma(Ref "x")(Ref "x"))) [])
   v = FClosure "x" (Soma(Ref "x")(Ref "x")) []
   e = EClosure (Soma(Valor 3)(Valor 4)) []
avaliar (Soma(Ref "x")(Ref "x")) [(x, EClosure (Soma(Valor 3)(Valor 4))):[]]
avaliarExpBin (Ref "x") (Ref "x") (+) [(x, EClosure (Soma(Valor 3)(Valor 4)))]
        env = [(x, EClosure (Soma(Valor 3)(Valor 4)))]
  (VInt ve) = avaliar (Ref "x") [(x, EClosure (Soma(Valor 3)(Valor 4)))]
            = pesquisar x [(x, EClosure (Soma(Valor 3)(Valor 4)))]
            = avaliacaoStrict (EClosure (Soma(Valor 3)(Valor 4)) env)
            = avaliacaoStrict (avaliar Soma(Valor 3)(Valor 4) env)
            = avaliacaoStrict (VInt 7)
            = VInt 7
         ve = 7
       env' = [(x, EClosure (Valor 7)):(x, EClosure (Soma(Valor 3)(Valor 4)))]
  (VInt vd) = avaliar (Ref "x") [(x, EClosure (Valor 7)):(x, EClosure (Soma(Valor
            = pesquisar x [(x, EClosure (Valor 7) [] ):env]
            = avaliacaoStrict (VInt 7)
            = VInt 7
         vd = 7
VInt ((+) 7 7)
VInt 14
```

Referências

[1] Eelco Dolstra e Eelco Visser. "Building Interpreters with Rewriting Strategies". Em: (2002).

[2] Shriram Krishnamurthi. *Programming Languages: Application and Interpretation*. Brown University, 2006.

[3] Michael L. Scott. *Programming Languages Pragmatics*. 3rd. Morgan Kaufmann, 2009.