Implementação de Lazy Evaluation e Recursão em Haskell

Hélio Santana da Silva Júnior - 140142959
Jônatas Ribeiro Senna Pires - 140090983

26 de junho de 2017

1 Introdução

Este documento apresenta uma implementação, em *literate Haskell*, de uma linguagem de programação funcional minimalista utilizando a estratégia de avalição lazy evaluation (sharing), que posterga as avaliações de expressões até que elas sejam necessárias. O documento também apresenta uma nova versão desta linguagem com suporte a recurssão.

2 Visão geral da linguagem

A linguagem LFCFD suporta tanto expressões identificadas (LET) quanto identificadores e funções de alta ordem (com o mecanismo de expressões lambda) além de expressões recursivas e um condicional simples. A linguagem também possui suporte para escopo estático.

3 Definição da Árvore Sintática Abstrata

A implementação se baseia na definição de um módulo Haskell e alguns tipos auxiliares, como `Id` (um tipo sinomimo para uma string) e Env, que corresponde a um ambiente de valores, onde um valor pode ser um valor inteiro ou uma expressão lambda.

module *LFCFDLazy* **where**

type *Id* = *String*
type *Env* = [(*Id*, *ValorE*)]

Existem duas definições de tipos, `ValorE` e `Expressao`, onde o `ValorE` pode ser definido como um valor inteiro, um identificador com uma expressão (lambda) ou uma expressão simples. As expressão são definidas como um valor inteiro, expressões binárias (soma, subtração, multiplicação e divisão), expressões `Let`, aplicações de funções, recursão e `IF0`.

```
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
    | Recursao Id Expressao Expressao
    | IF0 Expressao Expressao Expressao
deriving (Show, Eq)
```

A função que avalia as expressões recebe uma expressão e um ambiente e retorna um valorE. A avaliação de expressões necessita retornar uma closure, levando em conta que a linguagem esta utlizando substituições postergadas. O closure mapeia cada variável livre da função dentro do contexto da própria função.

```
avaliar :: Expressao → Env → ValorE
```

Para o caso de um valor inteiro a avialção é trivial, o retorno é um VInt, definido pelo `ValorE`. No caso de uma referência, e utilizado a função pesquisar, que realiza uma pesquisa da variável de referência dentro do ambiente. A função pesquisar assim como as demais funções utilizadas dentro da função avaliar serão explicadas posteriormente neste documento.

$avaliar\ (Valor\ n)\ \_ = VInt\ n$
$avaliar\ (Ref\ v)\ env = pesquisar\ v\ env$

Em relação aos casos de expressões binárias a avalição e muito semelhante em todos os casos. É usada a função `avaliarExpBin` que recebe duas expressões, um operador e o ambiente.

$avaliar\ (Soma\ e\ d)\ \ \ env = avaliarExpBin\ e\ d\ (+)\ env$
$avaliar\ (Subtracao\ e\ d)\ env = avaliarExpBin\ e\ d\ (-)\ env$
$avaliar\ (Multiplicacao\ e\ d)\ env = avaliarExpBin\ e\ d\ (*)\ env$
$avaliar\ (Divisao\ e\ d)\ \ \ env = avaliarExpBin\ e\ d\ div\ env$

A avalição de uma expressao `Let` é traduzida como uma aplicação lambda. A expressão lambda e convertida em uma closure com uma variável e uma expressão.

$avaliar\ (Let\ v\ e\ c)\ \ \ env = avaliar\ (Aplicacao\ (Lambda\ v\ c)\ e)\ env$
$avaliar\ (Lambda\ a\ c)\ env = FClosure\ a\ c\ env$

O caso de uma aplicação é o primeiro caso mais complexo da avaliação. Primeiramente é realizada uma avaliação diferente no primeiro argumento da aplicação, onde se espera retornar um `valorE`. Essa avalição foi chamada de `avaliacaoStrict`, que será aprofundada neste documento. Em seguida o segundo argumento e passado como uma closure de apenas uma expressão e ambiente. Caso a `avaliacaoStrict` retorne uma `FClosure`, e retornado uma avalição do corpo da closure passando o `Id` e a `Eclosure` dentro do ambiente, caso contrario é retornado um erro de aplicação de função.

$avaliar\ (Aplicacao\ e1\ e2)\ env =$
  let
    $v = avaliacaoStrict\ (avaliar\ e1\ env)$
    $e = EClosure\ e2\ env$
  in case v **of** -- if v = (FC a c env')
    $(FClosure\ a\ c\ env') \rightarrow avaliar\ c\ ((a, e) : env')$
    $otherwise \rightarrow error$ `"Tentando aplicar uma expressao que nao eh uma funcao a`

O caso de uma expressão `IF0` é simples. Caso a condição seja igual a zero a primeira expressão, t, é avaliada, caso contrário a segunda expressão, `e`, é avaliada. Esse tipo de expressão é usada para a construção de expressões

recursivas. A expressao recursiva é avaliada de forma similar a uma aplicação de função, levando em conta sua semelhança estrutural com a expressao `Let`. A diferença é a atualização do ambiente, que deve ser atualizado em cada "laço"da recursão.

```
avaliar (IF0 condicao t e) env
  | avaliar condicao env ≡ VInt 0 = avaliar t env
  | otherwise = avaliar e env
avaliar (Recursao identificador valor corpo) env =
let
  v = avaliacaoStrict (avaliar valor env)
  e = EClosure corpo env
  newEnv = (lookupApp identificador v env) ++ env
in case v of
  (FClosure a c env') → avaliar c ((a, e) : newEnv)
  otherwise → error "Tentando aplicar uma expressao que nao eh uma funcao ano
```

Em relação as funções auxiliares usadas na avalicação, a primeira a ser abodada será a função `avaliacaoStrict`. Ela recebe um `valorE` e retorna outro `valorE`. O objetivo dessa função é reduzir o `Eclosure` a um `VInt` ou retornar apenas um `FClosure`.

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

Outra função utilizada na função de avalição é a função pesquisar. Essa função é usada na avalicação de uma referência e ela recebe um `Id` e um ambiente e retorna um `valorE`. Ela procura o `Id` dentro do ambeiente e retorna a `avaliacaoStrict` da expressao referente a esse `Id`.

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

A função `lookupApp` é utilizada na avalição de uma expressão recursiva e ela é responsável pela atualização do ambiente no decorrer da avalicação

da recurssão. Caso seja passado um ambiente vazio, a função irá retornar uma lista com o idenditifcador e o valor passados como parâmetro. Caso o identificador seja igual ao `Id` da cabeça do ambiente, a função vai retornar uma lista vazia.

$$
\begin{aligned}
&\mathit{lookupApp} :: \mathit{Id} \to \mathit{ValorE} \to \mathit{Env} \to \mathit{Env} \\
&\mathit{lookupApp}\ \mathit{identificador}\ \mathit{valor}\ [\,] = [(\mathit{identificador}, \mathit{valor})] \\
&\mathit{lookupApp}\ \mathit{identificador}\ \mathit{valor}\ ((i, e) : \mathit{xs}) \\
&\mid \mathit{identificador} \equiv i = [\,] \\
&\mid \mathit{otherwise} = \mathit{lookupApp}\ \mathit{identificador}\ \mathit{valor}\ \mathit{xs}
\end{aligned}
$$

A última função utilizada é a função de avaliação de uma expressão binária, que avalia cada lado da expressão levando em consideração seus operadores.

$$
\begin{aligned}
&\mathit{avaliarExpBin} :: \mathit{Expressao} \to \mathit{Expressao} \to (\mathit{Int} \to \mathit{Int} \to \mathit{Int}) \to \mathit{Env} \to \mathit{ValorE} \\
&\mathit{avaliarExpBin}\ e\ d\ op\ env = \mathit{VInt}\ (op\ ve\ vd) \\
&\textbf{where} \\
&\quad (\mathit{VInt}\ ve) = \mathit{avaliacaoStrict}\ (\mathit{avaliar}\ e\ env) \\
&\quad (\mathit{VInt}\ vd) = \mathit{avaliacaoStrict}\ (\mathit{avaliar}\ d\ env)
\end{aligned}
$$

4 Conclusao

O presente trabalho consistiu no desenvolvimento de uma linguagem mínima no intuito de abordar conceitos como Lazy evaluation (sharing) e expressões recursivas. Foi mostrado que a linguagem também dava suporte para funções lambdas, expressoes `Let` e `IF0`.