# Normal-order Evaluator of $\lambda$-terms
# Homework Assignment 3

Rostislav Horčík

April 7, 2021

## 1 Introduction

This homework assignment aims to implement an evaluator of $\lambda$-expressions in Haskell reducing a given $\lambda$-expression into its normal form following the normal order evaluation strategy. As a side effect, this homework assignment will also help you to consolidate your knowledge on $\lambda$-calculus. The $\lambda$-expressions are going to be represented as instances of a suitably defined Haskell data type. So you will practice how to work with such types, in particular how to make it an instance of the Show class and how to use pattern matching with them.

The interpreter should be implemented as a Haskell module called `Hw3`. Note the capital `H`. The module names in Haskell have to start with capital letters. As the file containing the module code must be of the same name as the module name, **all your code is required to be in a single file called `Hw3.hs`.** Your file `Hw3.hs` has to start with the following lines:

```haskell
module Hw3 where

type Symbol = String
data Expr = Var Symbol | App Expr Expr | Lambda Symbol Expr deriving Eq
```

The first line defines a module of the name `Hw3`. The names of variables in $\lambda$-terms are represented by instances of `String`. The second line just introduces a new name `Symbol` for the type `String` to distinguish visually when we deal with the variable names. The last line defines the data type representing $\lambda$-expressions. There are three data constructors: one for a variable, one for an application and one for $\lambda$-abstraction. The clause **`deriving Eq`** makes the data type instance of the `Eq` class so that it is possible to check whether two $\lambda$-expressions are equal or not.

## 2 Evaluator specification

First, make the data type `Expr` an instance of the `Show` class so that `ghci` can display $\lambda$-expressions. So you need to define the `show` function converting $\lambda$-expressions into a string. Once you type into the `ghci` prompt the following expressions, it should behave as follows:

```
> Var "x"
x
> App (Var "x") (Var "y")
(x y)
> Lambda "x" (Var "x")
(\x.x)
> App (Lambda "x" (Var "x")) (Var "y")
((\x.x) y)
> Lambda "s" (Lambda "z" (App (Var "s") (App (Var "s") (Var "z"))))
(\s.(\z.(s (s z))))
```

So the symbol $\lambda$ is displayed as `\`. Variables are displays as their names. Applications are displayed as `(e1 e2)` with a space separating expressions `e1,e2` and $\lambda$-abstractions as `(\x.e)`.

As a next step, your task is to implement a function `eval :: Expr -> Expr`. This function for a given input $\lambda$-expression returns its normal form if it exists. Moreover, it has to follow the normal order evaluation strategy. So to make a single step $\beta$-reduction, you have to identify the leftmost outermost redex and reduce it. Then you repeat this process until there is no redex.

To reduce a redex, you have to implement the substitution mechanism allowing to substitute a $\lambda$-expression for all the free occurrences of a variable in another $\lambda$-expression. This mechanism has to deal with name conflicts as you know from the lecture on $\lambda$-calculus. One possibility how to safely do that is the following recursive definition:

$$\begin{aligned}
x[x := e] &= e \\
y[x := e] &= y \quad \text{if } y \neq x \\
(e_1\, e_2)[x := e] &= (e_1[x := e]\, e_2[x := e]) \\
(\lambda x.e')[x := e] &= (\lambda x.e') \\
(\lambda y.e')[x := e] &= (\lambda y.e'[x := e]) \quad \text{if } y \neq x \text{ and } y \text{ is not free in } e \\
(\lambda y.e')[x := e] &= (\lambda z.e'[y := z][x := e]) \quad \text{if } y \neq x \text{ and } y \text{ is free in } e\text{; for a fresh variable } z
\end{aligned}$$

The last case deals with the name conflicts, i.e., it uses $\alpha$-conversion. As $y$ is free in $e$ in this case, it could become bound after the substitution in $e'$. Thus we rename $y$ in $\lambda y.e'$ to a new fresh variable $z$, i.e., we compute $e'[y := z]$ and then replace the variable in the $\lambda$-abstraction to $\lambda z.e'[y := z]$. Then we can continue and recursively substitute $e$ for $x$ in $e'[y := z]$. So follow the above recursive definition in your code.

To obtain an unlimited number of fresh variables, define an infinite stream of variable symbols `symbols :: [Symbol]`, e.g.,

```
"a0","a1","a2",...
```

Your code has to track which symbols were already used and once a new fresh symbol is needed, the next one is used. To track the used symbols, it suffices to keep an index into the stream `symbols`. This index is a state of your computation. As Haskell is a purely functional language, you cannot have a state and update it when necessary. Instead, you need to include the index into signatures of your functions similarly, as we did it in the exercise from Lab 9 where we implemented a function indexing nodes of a binary tree. It is a good exercise to understand this technique because it will help you to grasp the idea behind the state monad. The state monad (and monads in general) is a useful abstraction allowing to get rid of the state tracking in purely functional languages like Haskell.

## 3 Test cases

Below you find several public test cases. If the $\lambda$-expression is already in its normal form, the `eval` function just returns its input.

```
> eval (App (Var "x") (Var "y"))
(x y)
> eval (Lambda "x" (Var "x"))
(\x.x)
```

If it is reducible, it returns its normal form. For instance $(\lambda x.x)y$ is reduced to $y$:

```
> eval (App (Lambda "x" (Var "x")) (Var "y"))
y
```

Consider the expression $(\lambda x.(\lambda y.(x\,y))y$. The reduction gives $(\lambda y.(x\,y))[x := y]$. By the definition of substitution, we have to rename $y$ to $a0$ and then substitute $y$ for the free occurrences of $x$, i.e.,

$$\begin{aligned}(\lambda y.(x\,y))[x := y] &= (\lambda a0.(x\,y)[y := a0][x := y])\\ &= (\lambda a0.(x\,a0)[x := y])\\ &= (\lambda a0.(y\,a0))\end{aligned}$$

```
> eval (App (Lambda "x" (Lambda "y" (App (Var "x") (Var "y")))) (Var "y"))
(\a0.(y a0))
```

Consider the $\lambda$-expression $(\lambda x.(\lambda y.y))y$. The reduction leads to $(\lambda y.y)[x := y]$. As $y$ is free, we introduce by the definition a fresh variable $a0$ instead of $y$ obtaining $\lambda a0.y[y := a0] = \lambda a0.a0$. Then we compute $\lambda a0.a0[x := y] = \lambda a0.a0$.

```
> eval (App (Lambda "x" (Lambda "y" (Var "y"))) (Var "y"))
(\a0.a0)
```

Consider the $\lambda$-expression $(\lambda x.(\lambda y.(\lambda z.((xy)z))))(y\,z)$. As $y$ and $z$ are free in $(y\,z)$ we have to rename them in $\lambda x.(\lambda y.(\lambda z.((xy)z)))$ obtaining $\lambda a0.(\lambda a1.(((y\,z)\,a0)\,a1))$.

```
ex = App (Lambda "x"
           (Lambda "y"
             (Lambda "z" (App (App (Var "x") (Var "y")) (Var "z")))))
         (App (Var "y") (Var "z"))

> eval ex
(\a0.(\a1.(((y z) a0) a1)))
```

To write more complex test cases, you can define subexpressions and then compose more complex ones out of them. For instance, to test that $S1$ reduces to 2:

```
one = Lambda "s" (Lambda "z" (App (Var "s") (Var "z")))
suc = Lambda "w"
        (Lambda "y"
          (Lambda "x"
            (App (Var "y")
                 (App (App (Var "w") (Var "y"))
                      (Var "x")
                 )
            )
          )
        )

> eval (App suc one)
(\y.(\x.(y (y x))))
```

One more test case using the definition $one = \lambda s.(\lambda z.(s\,z))$. Consider the $\lambda$-expression

$$(\lambda z.one)(s\,z) = (\lambda z.(\lambda s.(\lambda z.(s\,z))))(s\,z).$$

It reduces to $\lambda a0.(\lambda z.(a0\,z))$. As $s$ is free in $(s\,z)$, the bound occurrence of $s$ in $one$ is renamed. But $z$ is not renamed because by the defition of substitution we have $\lambda z.(a0\,z)[z := (s\,z)] = \lambda z.(a0\,z)$.

```
> eval (App (Lambda "z" one) (App (Var "s") (Var "z")))
(\a0.(\z.(a0 z)))
```