CS 558 Homework 3

Arjun Krishna Babu

7 November 2017

Part (a)

Define a haskell type `Term` to represent the terms of the untyped lambda calculus.

```
data Term = Var String       -- variable
  | Lambda String Term       -- abstraction
  | Application Term Term    -- application
  deriving (Show, Eq)
```

The `String` in `Var` and `Lambda` is where you store the *name* of the variable.

For example, λ `x.x` would be represented as `Lambda “x” (Var “x”).`

```
*Main> :t Lambda "x" (Var "x")
Lambda "x" (Var "x") :: Term
```

Part (b)

Define a haskell function `subst`, such that `subst x t1 t2` implements the capture avoiding substitution operation `[x ↦ t1]t2`

```
-- [x -> s]y
subst :: Term -> Term -> Term -> Term

-- variable case
subst (Var x) s (Var y) = if x == y
  then s
  else (Var y)

-- application case
subst x s (Application t1 t2)  = Application (subst x s t1) (subst x s t2)

-- abstraction case
subst a@(Var x) b c@(Lambda y t) = if x == y
  then c
  else Lambda y (subst a b t)
```

Examples

I. Variable Case

Let's first define our variables and lambda terms:

```
*Main> x = Var "x"
*Main> y = Var "y"
*Main> r = Lambda "r" (Var "g")
```

Now let's apply `subst`:

```
*Main> subst x r y
Var "y"

*Main> subst x r x
Lambda "r" (Var "g")
```

This is as expected based on the following rules:

- `[x ↦ r]x = x`
- `[x ↦ r]x = y, if y ≠ x`

II. Abstraction Case

Here is a case when the variable and variable in lambda term are equal. ie., `[x ↦ r](λx.t)`

```
*Main> x = Var "x"
*Main> r = Lambda "r" (Var "g")
*Main> xx = Lambda "x" (Var "m")
*Main> subst x r xx
Lambda "x" (Var "m")
```

Here is a case that represents `[x ↦ r](λy.t)`

```
*Main> x = Var "x"
*Main> r = Lambda "r" (Var "g")
*Main> dd = Lambda "d" (Var "n")
*Main> subst x r dd
Lambda "d" (Var "n")
```

Part (c)

Define a Haskell function `isValue`, that returns

- `True`, if the lambda term represented by `t` is a value
- `False`, otherwise

Values have the structure (λ `x.t`)

The function `isValue` defined below works by examining the structure of the argument passed to it:

```
isValue :: Term -> Bool
isValue (Lambda _ _) = True
isValue _ = False
```

Example

```
*Main> isValue (Lambda "x" (Var "g"))
True

*Main> isValue (Var "e")
False
```

Part (d)

Define a haskell function `eval1` that implements single-step reduction of lambda calculus terms encoded as values of type `Term`.

```
eval1 :: Term -> Maybe Term

-- E_APPABS: (Lx.t)v -> [x->v]t
eval1 (Application (Lambda x t) v2) = if isValue v2
  then Just (subst (Var x) v2 t)
  else Nothing

eval1 (Application t1 t2) = if isValue t1
  then case (eval1 t2) of   -- E_APP2
       Just t -> Just (Application t1 t)
       otherwise -> Nothing
  else case (eval1 t1) of   -- E_APP1
       Just t -> Just (Application t t2)
       otherwise -> Nothing

eval1 _ = Nothing
```

Part (e)

Define a haskell function `eval` that recursively calls `eval1` to evaluate a lambda calculus term as many times as possible.

```
eval :: Term -> Term
eval t = case (eval1 t) of
  Just t' -> eval t'
  Nothing -> t
```

Part (f)

Demonstrate the use of your `eval` function to reduce the following untyped lambda-term to its normal form

```
(λx . x x)(λy . y)
```

Let's first manually do this calculation in lambda calculus to figure out the expected result:

```
(λx.x x)(λy.y) = (λy.y) (λy.y)
   = (λy.y)
```

Evaluating the same using the definitions defined above yields:

```
-- define (λx.x x)
*Main> xt = Lambda "x" (Application (Var "x") (Var "x"))

-- define (λy.y)
*Main> yt = Lambda "y" (Var "y")

-- apply the eval function
*Main> eval (Application xt yt)
Lambda "y" (Var "y")
```

This is equivalent to the expected result `(λy.y)`

References

- Wikipedia: Lambda calculus
  https://en.wikipedia.org/wiki/Lambda_calculus