CSCI 4430 Programming Languages

Homework 6: An Interpreter for the Lambda calculus in Haskell

Due: Tuesday November 24 @ 1:59pm

Submission Instructions

This is an individual assignment. Just as with all other homework, submitted work should be your own. Course staff runs plagiarism detectors and will treat excessive similarities between submissions as evidence of cheating. Submit in Submitty for autograding. Your Haskell file must be named `Interpreter.hs`. Submitty compiles your Interpreter module using `The Glorious Glasgow Haskell Compilation System, version 8.0.2` and links it with various tests that call your functions.

Getting Started with Haskell

Download and install the Glasgow Haskell Compiler: `https://www.haskell.org/ghc`.

You can run GHC in interactive mode by running `ghci` from the command line. Type the following functions and save them in a file called `fun.hs`:

```
apply_n f n x = if n == 0 then x
                else apply_n f (n-1) (f x)

plus a b = apply_n ((+) 1) b a

mult a b = apply_n ((+) a) b 0

expon a b = apply_n ((*) a) b 1
```

Run `ghci` then load the file by typing

```
:l fun.hs
```

and call these functions, e.g., `plus 2 3`. You can reload the file after making a change

```
:r
```

An Applicative Order Interpreter for the Lambda Calculus

Now, we get to the actual homework. In lecture, we wrote a normal order interpreter for the Lambda Calculus. It terminated when it returned an answer in *Weak Head Normal Form*. In this problem we will write an applicative order interpreter that yields an answer in *Normal Form* (i.e., the expression cannot be further reduced).

Problem 1. Applicative order interpreter

Your first task is to write the interpreter in the style we gave in lecture. Please include your answer in comments in your Haskell file under heading Problem 1.

Problem 2. Coding the applicative order interpreter in Haskell

Next, you will code the interpreter in Haskell. A lambda expression is of the following form, as we discussed in class:

```
data Expr =
          Var Name            --- a variable
          | App Expr Expr     --- function application
          | Lambda Name Expr --- lambda abstraction
deriving
   (Eq,Show) --- the Expr data type derives from built-in Eq and Show classes,
             --- thus, we can compare and print expressions

type Name = String --- a variable name
```

You are required to implement the following functions:

Free variables. As a first step, write the function

```
freeVars :: Expr -> [Name]
```

It takes an expression `expr` and returns the list of variables that are free in `expr` without repetition. For example, `freeVars (App (Var "x") (Var "x"))` yields `["x"]`.

Generating new names. Next, generate fresh variables for a list of expressions by making use of the infinite list of positive integers `[1..]`. Write a function

```
freshVars :: [Expr] -> [Name]
```

It takes a list of expressions and generates an (infinite) list of variables that are *not* free in any of the expressions in the list. For example, `freshVars [Lambda "1_" (App (Var "x") (App (Var "1_") (Var "2_")))]` yields the *infinite* list `[1_,3_,4_,5_,..]`. Remember, you will leverage `[1..]` to implement `freshVars`.

Substitution. Next, write the substitution function

```
subst :: (Name,Expr) -> Expr -> Expr
```

All functions in Haskell are *curried*: i.e., they take just one argument. The above function takes a variable `x` and an expression `e`, and returns a function that takes an expression `E` and returns `E[e/x]`. **Important:** `subst` must implement the algorithm we gave in class! Specifically, when performing a substitution on a lambda expression `λy.E1`, where `y≠x`, `subst` must always replace parameter `y` with the next fresh variable from the list of `freshVars`, even if `y` is not free in `e` and it would have been safe to leave it as is. This is necessary for autograding on Submitty.

A single step. Now write a function to do a single step of reduction:

```
appNF_OneStep :: Expr -> Maybe Expr
```

where the built-in `Maybe` type is defined as follows:

```
data Maybe a =
      Nothing
      | Just a
```

`appNF_OneStep` takes an expression `e`. If there is a redex available in `e`, it picks the correct applicative order redex and reduces `e`. (Note that in applicative order we are pursuing the leftmost, innermost strategy as follows. Pick the leftmost redex *R*; if there are nested (inner) redexes within *R*, pick the leftmost one, and so on, until we reach a redex without nested ones.) If a reduction was carried out, resulting in a new expression `expr'`, `appNF_OneStep` returns `Just expr'`, otherwise it returns `Nothing`.

Repetition. Finally, write a function

```
appNF_n :: Int -> Expr -> Expr
```

Given an integer `n` and an expression `e`, `appNF_n` does `n` reductions (or as many as possible) and returns the resulting expression.

- Write all functions described above, `freeVars, freshVars, subst, appNF_OneStep`, and `appNF_n` and include them in your `Interpreter.hs` file.
- You are allowed to use all features of Haskell and import modules you find useful.
- Just as with your Scheme homework, do comment your code! Write comments in the same style although there is one notable difference, namely that in Haskell type signatures are statically checked. Include type signatures for every function you write even though Haskell can infer those signatures.

Minimal starter code is provided here.

Notes on grading: This homework is worth a total of 50 points: 40 points will be awarded for functional correctness by the autograder and 10 points will be awarded for quality and completeness of Problem 1 and comments.

Errata

None yet. Check the Announcements on Submitty regularly.