Homework 5

Posted Monday February 29, Due Monday March 18

50 points

1. Abstract Interpretation (12 points)

1.1. **Properties of Galois Connections.** (6 pts) Let $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \leq)$ be an arbitrary Galois Connection. Prove that α is monotone, i.e., for any two elements c_1, c_2 of concrete domain C, we have $c_1 \subseteq c_2 \Rightarrow \alpha(c_1) \leq \alpha(c_2)$. Hint: use the expansivity property we discussed in class.

1.2. **Properties of Galois Connections.** (6 pts) Let $(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \leq)$ be an arbitrary Galois Connection. Prove that γ is monotone, i.e., for any two elements a_1, a_2 of abstract domain A, we have $a_1 \leq a_2 \Rightarrow \gamma(a_1) \subseteq \gamma(a_2)$. Hint: use the contractivity property.

2. Applicative Order Interpreter for the λ-Calculus (38 points)

(Adapted from the MIT Program Analysis Course.)

2.1. **Getting Started with Haskell.** (0 pts) Download and install the Glasgow Haskell Compiler:

`https://www.haskell.org/ghc/`

You can run GHC in interactive mode by running `ghci`. Type the following and save it in a file `p1.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
```

then run `ghci` and load the file by typing

```
:l p1.hs
```

You can reload the file after a change on disk by typing

```
:r
```

Try different calls in interactive mode, e.g., `expon 3 3`, etc.

Now, let's get to the real homework. In lecture, we wrote two interpreters for the λ-Calculus: a call-by-name interpreter and a call-by-value interpreter. Both interpreters terminated when they returned an answer in *Weak Head Normal Form (WHNF)*. In this problem we will write a *Applicative-Order* interpreter that yields a value in *Normal Form* (i.e., the expression cannot be reduced anymore).

2.2. **Step-by-step Reduction.** (5 pts) Consider the following term:

$$(\lambda x.\lambda y.x)(\lambda z.(\lambda x.\lambda y.x)\ z\ ((\lambda x.z\ x)(\lambda x.z\ x)))$$

Give the **first 2 reduction steps** for *normal order* and *applicative order* each. Remember, in normal order we choose the leftmost redex. 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, which we reduce.

You will code in Haskell an *applicative order* interpreter reducing to *normal form*. An 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
```

2.3. **Free variables.** As a first step, write the funciton

`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"]`.

2.4. **Generating fresh 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_,..]`.

2.5. **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]`. **Note**: `subst` must implement the substitution algorithm we gave in class! Specifically, when performing a substitution on a lambda expression λ`y.E1`, where `y` $\neq$ `x`, `subst` always replaces 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.

2.6. **A single step.** Now write a function to do a single step of *applicative order* 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 `expr`. If there is a redex available in `expr`, it picks the correct applicative order redex and reduces `expr`. If a reduction was carried out, resulting in a new expression `expr1`, `appNF_OneStep` returns `Just expr1`, otherwise it returns `Nothing`.

2.7. **Repetition.** Now write a function

```
appNF :: Int → Expr → Expr
```

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

When you are done, upload your files in Submitty. Upload a pdf file with solutions for 1.1-1.2 and 2.2. Upload file `Interpreter.hs` with the interpreter code, i.e., the implementation of functions described in 2.3-2.7. `Interpreter.hs` will be autograded and has to start with

```
module Interpreter where
import ...
...
```