Fakultät für Informatik und Mathematik Summer semester 2023
Prof. Dr. Martin Griebl

Exercises for Lecture: Functional Programming
Exercise Sheet 11 (λ-Calculus, Strictness)

Problem 1 (λ-Calculus)

An application $(\lambda x.\, E)\, A$ is evaluated (i.e., a β-reduction is performed) by substituting A for x in E: $(\lambda x.\, E)\, A \to_\beta E\, [x := A]$. Evaluate the following λ-expressions as far as possible (including the evaluation of arithmetic expressions). Underline the redex of each β-reduction and state the substitution explicitly (i.e., give the intermediate step $E\, [x := A]$ before carrying out the substitution).

(a) $\big(\lambda f.\, f\,(x{+}1)\big)\;(\lambda x.\, 2{\cdot}x)$

(b) $\big(\lambda y.\, (\lambda x.\, x{\cdot}y)\;(y+1)\big)\; x$

(c) $(\lambda f.\, f\, f)\;\big(\lambda x.\, (\lambda y.\, x)\big)\;(\lambda z.\, z)$

Problem 2 (Fixed-Point Operator)

The lecture introduced the fixed-point operator Y, which satisfies $Y\; f =_\beta f\;(Y\; f)$, but not $Y\; f \twoheadrightarrow_\beta f\;(Y\; f)$. In other words, it is not possible to go from $Y\; f$ to $f\;(Y\; f)$ with β-reductions alone, on has to perform one β-reduction "backwards", i.e, formally, Y has the property that there is a λ-term t which satisfies $Y\, f \twoheadrightarrow_\beta t$ und $f\;(Y\; f) \to_\beta t$.

Show (by performing the necessary β-reductions) that the λ-term $Z = V\; V$ with $V = \big(\lambda z\; x.\, x\,(z\; z\; x)\big)$ has the property $Z\; f \twoheadrightarrow_\beta f\;(Z\; f)$.

(Y was discovered Haskell B. Curry, Z by Alan Turing.)

Problem 3 (Enriched λ-Calculus)

Let the following abstract syntax for λ-expressions be given (cf. the files `Syntax.hs`, `Reduce.hs` und `Main.hs` in Stud.IP):

```
data LExp = V String                  -- variable
          | CInt Int                  -- Int constant
          | CBool Bool                -- Bool constant
          | LExp :@: LExp             -- application
          | L String LExp             -- λ-abstraction
          | If LExp LExp LExp         -- If c t e  means  if c then t else e
          | Y LExp                    -- fixed point operator
          | Prim Op [LExp]            -- predefined operator
          deriving (Show)
```

This syntax has been enriched (compared to the simple λ-calculus) with constructors for integer and boolean constants, if-then-else, `Y` and primitive operations.

Complete the definitions in the modules `Synatax` and `Reduce`. Implement the functions

(a) `fv, bv :: LExp -> [String]` to determine the free and bound variables in an expression,

(b) `occursNot :: [String] -> String` to generate a fresh name,

(c) `substitute :: String -> LExp -> LExp -> LExp` for substitution, and

(d) `red_Redex_LMHead :: LExp -> Maybe LExp` to perform a reduction step towards weak head normal form (WHNF). Use a suitable definition for a reduction step for `Y` und `If`.

Test your implementation with module `Main`. Calling `main` should output `CInt 120` as the reduction result.

Problem 4 (Strictness of `foldl`)

Let us consider the function `foldl` again:

```
foldl :: (a -> b -> a) -> a -> [b] -> a
foldl f x []     = x
foldl f x (y:ys) = foldl f (f x y) ys
```

We want to study how `foldl` behaves when a strict function `f` is used. Start GHCi for this exercise with `ghci +RTS -K10m -RTS` to limit the stack to 10MB.

(a) Evaluate in GHCi `foldl (+) 0 [1..n]` for different values of n; with the above setting for the run-time stack, a stack overflow should occur around n=320000 (on x86_64) or n=640000 (on x86). How is that possible although `foldl` is defined by a tail recursion?

(b) Could the compiler, at least in theory, avoid the problem, if it had access to more information (e.g., about `(+)`)?

(c) The problem can be remediated by putting strictness annotations in the definition (using `seq`). Implement a function `foldlStrict`, which uses `seq` to realize a reduction with constant stack consumption. Test with

```
foldlStrict (+) 0 [1..10^7].
```

Note: The semantics of x `` `seq` `` y is to evaluate x, then return y. x is evaluated as far as is needed to determine which is the outermost constructor in the expression, i.e., x is evaluated to weak head normal form. The arguments of the constructor are not evaluated.

Examples:

```
Prelude> let x = error "bar"  in  x `seq` 5
*** Exception: bar
Prelude> let x = error "bar"  in  5
5
Prelude> let xs = [ 1, 2, 3 `div` 0 ]  in  xs `seq` 42
42
(There is no excption here because xs = (:) 1 [ 2, 3 `div` 0 ]
 and constructor arguments are not evaluated.)
```

Due date: Tuesday, July 4, 2023 at 16:00

Upload your solutions to the `Submissions` folder for this exercise in Stud.IP.