P. Thiemann Wintersemester 2022/23

Functional Programming

https://proglang.informatik.uni-freiburg.de/teaching/functional-programming/2022/

Exam Sheet

March 13-15, 2023

How to do this exam

- You can earn 50 points from the questions on this sheet.
- You can earn 150 points from the programming task.
- You need $\geq$ 100 points to pass the exam and you must earn $\geq$ 20 points from this sheet. Otherwise you are free to choose.
- Do not change the names and types of the functions as given on this sheet and in the template files supplied.

Question 1 (Terms, 20 points, file: Question1.hs)

In the lecture, we defined signatures and terms. Consider the following datatypes for representing signatures where function symbols are identified by `Ident` and terms over an empty set of variables.

```
import qualified Data.Map as M

type Ident     = Char
data Term      = Term { symbol :: Ident, subterms :: [Term] }
type Signature = M.Map Ident Int
```

(a)
```
parseTerm :: Parser Char Term
```

Write a parser that reads a string as a "raw term" according to the grammar of terms (derived from symbol T, without checking the correct use of symbols). Terminal symbols in the grammar are either literal strings in single quotes (e. g. `','`) or regular expressions in double quotes (e. g. `"[a-zA-Z]"`, which denotes an ASCII letter).

$$T ::= \texttt{"[a-zA-Z]"}\ \texttt{'('}\ A\ \texttt{')'} \qquad A ::= \varepsilon \mid T\,B \qquad B ::= \varepsilon \mid \texttt{','}\ T\,B$$

(b)
```
type Pos = [Int]
arityCheck :: Signature -> Term -> Either Pos Term
```

Write an arity checker that checks that a raw term is well-formed according to a given signature. The implementation of the arity checker should use the `Either` monad to report errors in terms of a list of numbers that describes the location of the first error: `[]` for an error at the root of the term, `[0]` for the root of the first subterm, `[0,1]` for the root of the second subterm of the first subterm, and so on.

```
>> sig = M.fromList [('x', 0)]
>> arityCheck sig (Term 'x' [])
Right (Term 'x' [])
>> arityCheck sig (Term 'x' [Term 'x' []])
Left []
```

Question 2 (Substitution and reduction, 30 points, file: Question2.hs)

Consider the following datatypes for representing expressions in lambda calculus.

```haskell
import qualified Data.Set as S

type Ident = String
data Expr
  = Var Ident
  | Lam Ident Expr
  | App Expr Expr
```

(a) `free :: Expr -> S.Set Ident`

Define a function to calculate the set of free variables of a lambda expression (using the `Set` type from `Data.Set`).

(b) `fresh :: S.Set Ident -> Ident`

Define a function to return a string that is not a member of a given set of strings. The function does not have to be efficient, but it must terminate for all inputs.

(c) `subst :: Expr -> Ident -> Expr -> Expr`

Define capture-avoiding substitution for arbitrary lambda expressions. `subst t' x t` substitutes `t'` for `x` in `t`.

```
>> ex0  = Lam "y" (App (Var "x") (Var "y"))
>> ex0' = Var "y"
>> subst ex0' "x" ex0
Lam "x0" (App (Var "y") (Var "x0"))
```

(d) `tryBeta :: Expr -> Maybe Expr`

Define a function that locates the leftmost beta redex in a lambda expression and reduces it. To find the leftmost redex, traverse the expression in preorder. As no such redex may exist, you should write this function using the `Maybe` applicative.

```
>> ex1 = App (Lam "x" (Var "x")) (Var "y")
>> ex2 = App (Var "x") ex1
>> ex3 = Lam "x" ex2
>> tryBeta ex1
Just (Var "y")
>> tryBeta ex2
Just (App (Var "x") (Var "y"))
>> tryBeta ex3
Just (Lam "x" (App (Var "x") (Var "y")))
```