Functional Programming

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

Exam Sheet

March 04-06, 2024

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 ≥ 100 points to pass the exam and you must earn ≥ 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 file supplied. You may add further imports as long as they do not require adding new files.

Consider the following datatype to represent arithmetic expressions for values of type `Double`:

```
type Ident = String

data Expr
  = Var Ident Integer           -- x^i
  | Con Double
  | Neg Expr                    -- - e
  | Add Expr Expr               -- e1 + e2
  | Rec Expr                    -- 1 / e
  | Mul Expr Expr               -- e1 * e2
```

Expressions comprise variables with an integer exponent, constants, negation, addition, reciprocal, and multiplication. There are two exercise questions for this datatype on the following pages.

Question 1 (Applicatives, 20 points, file: Questions.hs)

(a)
```
data EnvReader env a = ...

instance Applicative (EnvReader env) where
    ...

runEnvReader :: EnvReader env a -> env -> Maybe a
runEnvReader = undefined
```

Define an applicative `EnvReader env a = env -> Maybe a` that provides every subcomputation with a fixed environment `env` and returns its value of type `a` wrapped in a `Maybe`. Provide a function `runEnvReader :: EnvReader env a -> env -> Maybe a` to run such an applicative.

(b)
```
type Env k v = [(k, v)]

getKey :: Env k v -> k -> Maybe v
getKey = undefined
```

The type `Env k v` represents a finite mapping from some key type `k` to values drawn from type `v` by a suitable list. Implement a function `getKey` that applies the mapping to a key.

(c)
```
eval :: Expr -> EnvReader (Env Ident Double) Double
eval = undefined
```

Define an evaluation function for the expression datatype `Expr` that takes an expression `e` and an environment that maps identifiers to `Double` values and returns the value of `e` or `Nothing` if any variable in `e` is not defined in the environment or if any division by zero is attempted. Write this function in applicative style as indicated by the signature.

Examples

```
λ> runEnvReader (eval (Con 4711)) []
Just 4711.0
λ> runEnvReader (eval (Var "x" 1)) []
Nothing
λ> example0 = Add (Mul (Con 2) (Var "x" 1)) (Con 1)
λ> runEnvReader (eval example0) [("x", 20.5), ("y", 10)]
Just 42.0
```

Question 2 (Simplification, 30 points, file: Questions.hs)

(a)
```
simplify :: Expr -> Expr
simplify = undefined
```

Define a function `simplify` that simplifies an expression as follows:

- Evaluation of the simplified expression should yield the same value as the original expression for every environment that defines all variables in the original expression.
- Simplification should always terminate; do not use commutativity!
- (5 points) Any expression without variables should be simplified to a constant. A division by zero should not be executed.
- (5 points) Propagate `Neg` and `Rec` to the outside as much as possible; adjacent `Neg` should cancel out; adjacent `Rec` should cancel out unless doing so changes the definedness of the expression.
- (5 points) Apply the arithmetic simplification laws for multiplication.
- (5 points) Apply the arithmetic simplification laws for addition.

Examples

```
λ> simplify $ Add (Neg (Mul (Con 56) (Neg (Neg (Con (-88)))))) (Con (-77))
Con 4851.0
λ> simplify $ Add (Neg (Rec (Con 0))) (Mul (-1) (Con (-77)))
Neg (Add (Rec (Con 0.0)) (Con 77.0))
λ> simplify $ Mul (Add (Var "x" 1) (Var "y" 1)) (Add (Var "x" 1) (Var "y" 1))
Add (Var "x" 2) (Add (Mul (Var "x" 1) (Var "y" 1))
                (Add (Mul (Var "y" 1) (Var "x" 1)) (Var "y" 2)))
```

(b)
```
genExpr :: Gen Expr
genExpr = undefined

genEnv :: Expr -> Gen (Env Ident Double)
genEnv = undefined
```

Use QuickCheck to define a define a test harness for the simplifier and use it to demonstrate the thesis that *simplification does not change the value of an expression.*

You will have to provide a generator for expressions as well as a generator for environments. Your generated environments should depend on a (generated) expression so that they do not contain bindings for variables that do not occur in the expression. Formulate the testing property so that it supports the thesis.