Haskell: Type Classes and Monads

Introduction

All the exercises below consider (variants of) the following ADT for simple expressions:

```
data Expr a = Const a | Sum (Expr a) (Expr a) | Mul (Expr a) (Expr a)
```

Exercise 1

Define a recursive evaluation function `eval` for expressions. Test the function on a couple of simple expressions. For example,

eval (Sum (Mul (Const 2) (Const 3)) (Const 4))

should evaluate to `10`.

- **Goal:** Warming up!
- **Expected output:** A function `eval` that recursively evaluates an expression.

Exercise 2

Enrich the above expressions with a new constructor `Div (Expr a) (Expr a)` and write an evaluation function `safeEval` for these extended expressions, interpreting `Div` as integer division. Test the new function with some expressions.

Hint: Function `safeEval` must be partial, since division by zero is undefined, and thus it must return a `Maybe` value.

- **Goal:** First steps with partial functions.
- **Expected output:** A function `safeEval` that recursively evaluates extended integer expressions.

Exercise 3

Define an instance of the constructor class `Functor` for the expressions of Exercise 1, in order to be able to `fmap` over trees. A call to `fmap f e` (where `e :: Expr a` and `f :: a -> b`) should return an expression of type `Expr b` obtained by replacing all the `Const v` nodes in `e` with `Const (f v)`.

- **Goal:** Experimenting with constructor classes.
- **Expected output:** An instance `Functor Expr`, as requested.

Exercise 4

Propose a way to define an instance `Foldable Expr` of the class constructor `Foldable`, by providing a function to fold values across a tree representing an expression.

Hint: Consult Hoogλe to discover the "Minimal complete definition" of `Foldable`. Several solutions are possible.

- **Goal:** Experimenting with the `Foldable` constructor class, and understanding Haskell documentation.

- **Expected output:** An instance `Foldable Expr`, as requested.

Exercise 5

Consider the following definition of variables for the expressions of Exercise 1:

```
data Var = X | Y | Z
data Expr a = ... | Id Var
```

First, define a function `subst` that takes a triple `(x, y, z)` of expressions, interpreted as the values of `X`, `Y`, `Z` respectively, and an expression and produces a new expression where the variables are substituted with the corresponding expressions.

Next define functions `eval` and `recEval`. The **partial** function `eval`, applied to an expression `e`, returns its value if `e` does not contain variables, and `Nothing` otherwise. Function `recEval` takes as arguments a triple of expressions `(x, y, z)` and an expression `e`, and evaluates `e` replacing variables with the corresponding expressions when needed.

Finally, compare the effect of applying function `recEval` and `subst.eval` to a triple of expressions `(x, y, z)` and an expression `e`. Do they always deliver the same result?

- **Goal:** Experiment a little more with partial function.
- **Expected output:** An implementation of the `subst`, `eval` and `recEval` functions.

Exercise 6

Write an instance of `Show` that allows to print expressions (with parenthesis!).

Hint: Take a look at the doc of Show.

- **Goal:** Giving another try to type classes.
- **Expected output:** An instance of `Show` of `Expr`.

Exercise 7

Consider the `eval` function of Exercise 1. Exploiting the IO monad, Write two new versions of `eval`:

1. `evalPrint`, that directly prints the final result of the expression under evaluation
2. `evalPrintSub`, that also prints all the intermediate results

Hint: Take a look at the IO Monad documentation.

- **Goal:** Experimenting with the I/O monad.
- **Expected output:** The two requested functions.

Author: Andrea Corradini & Matteo Busi

Created: 2019-11-22 Fri 00:23

Validate