Programming Paradigms 2025
Session 10: Functors

Problems for solving and discussing

Hans Hüttel

11 November 2025

Problems that we will definitely talk about

1. ***(20 minutes)***

   The type of unbounded trees UTree is given by

   ```
   data UTree a = Node a [UTree a]
   ```

   Define an instance of Functor for UTree.

2. ***(15 minutes)***

   Let r be some given type. The function type constructor Arrow−r is defined such that Arrow−r a will be (r −> a).

   Define an instance of Functor for this type constructor.

   In order to test your solution, add the following at the start of the file containing your code:

   ```
   import qualified Prelude
   import Prelude hiding (Functor, fmap)
   ```

3. ***(15 minutes)***

   For the applicative functor for lists we have a definion of the "funny star" composition <*> on page 160. Give an alternative *recursive* definition of it that uses fmap.

4. ***(15 minutes)*** Use the fact that the list type can be seen as an applicative functor to define a function prodthree that takes three lists of numbers and computes the list of all products of triples of numbers in the list. As an example, prodthree [1,2,3] [4,5,6] [7,8,9] should give us the list

   ```
   [28,32,36,35,40,45,42,48,54,56,64,72,70,80,90,84,96,108,84,96,108,
   105,120,135,126,144,162]
   ```

   Hint: Somewhere a funny star keeps shining.

More problems to solve at your own pace

a) Here is a type declaration for simple expressions.

```
data Exp a = Var a | Val Integer | Add (Exp a) (Exp a) |
   Mult (Exp a) (Exp a) deriving Show
```

Show how do make this type into an instance of Functor.

When would it be useful to think of Exp a as a functor? Think of a good example!

b) Show how to make the type Exp from the previous problem into an instance of Applicative.

c) In order to solve this problem, you must already have a definition of Exp as an applicative functor from problem b. Assume the definitions

```
type Name = String

type Env = [(Name, Int)]

fetch :: Name -> Env -> Int
fetch x env = case lookup x env of
    Nothing -> error "invalid name"
    Just v -> v
```

Now use all of these definitions to define a function

```
eval :: Expr -> Env -> Int
```

that will, when given an expression e and an environment env, return the value of the expression, assuming that all variables in e are given values in env.