Script: Functors and friends

Headline

Generalzing list maps and folds to other types

Description

List processing with maps and folds is generalized to designated classes of types. In particular, we look at data types that can be thought of as modeling *containers*, and we treat them in a way similar to lists. This is possible, in particular, for maybe types and rose trees. To this end, we leverage the notions of functor and foldable type. The corresponding type classes rely on higher-kinded polymorphism. We also demonstrate the use of functors and foldable types for more domain-specific types such as companies, departments, and employees. (It turns out that we need to adjust the domain-specific types to the purpose of (salary) transformation and aggregation. Finally, we exercise maps and folds in an advanced example of bidirectional transformation. In passing, we also engage in the more general notion of applicative functors which allow for functorial computations to be sequenced (unlike plain functors),

Concepts

- Type classes
  - Functor
  - Foldable type (fold)
  - Applicative functor
- Data types used for illustration
  - Rose tree
  - Maybe type
- Other concepts at play
  - Higher-kinded polymorphism
  - Bidirectional transformation

Languages

- Language:Haskell

Features

- Feature:Total
- Feature:Cut

Contributions

- Contribution:haskellFunctorial: Functorial total and cut
- Contribution:haskellNonfunctorial: Reusable abstractions for salary access
- Contribution:haskellTree: Bidirectional transformations

Concept: Functor

Headline

A functional programming idiom for mapping over containers

Illustration

The term "functor" originates from category theory, but this will be of no further concern in this description. In functional programming, "functor" refers to a programming idiom for mapping over contains or compound data. Functors have been popularized by Language:Haskell.

In Haskell, functors are programmed and used with the help of the type class *Functor* which is parametrized by a type constructor for the actual container type:

```
class Functor f
  where
    fmap :: (a -> b) -> f a -> f b
```

The type constructor parameter *f* is the placeholder for the actual container type. The fmap function (for "functorial map") is the principle operation of a functor: parametrized by a function for mapping container elements of type *a* to elements of type *b*, it provides a mapping at the level of the container types, from *f a* to *f b*. Algebraically, the following properties are required for any functor (given in Haskell notation):

```
fmap id = id
fmap f . fmap g = fmap (f . g)
```

The following *Functor* instance turns lists into a functor:

```
instance Functor []
  where
    fmap = map
```

Thus, the folklore map function for list processing is a particular example of the notion of functorial map.

Here is another *Functor* instance turning the Maybe type constructor into a functor.

```
instance Functor Maybe
  where
    fmap _ Nothing = Nothing
    fmap f (Just x) = Just (f x)
```

See also the concept of rose trees for more complicated examples of functors.

Concept: Foldable type

Headline

A type for which a fold function can be defined

Illustration

Obviously, a fold function can be defined for lists. See also the concept of Maybe type for another simple example of a foldable type. See the concept of rose tree for a more powerful illustration of a foldables.

In Language:Haskell, there is a type class of foldable types:

```
class Foldable t
  where
    fold :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldl :: (a -> b -> a) -> a -> t b -> a
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl1 :: (a -> a -> a) -> t a -> a
```

The members *foldr* and *foldl* generalize the function signatures of the folklore fold functions for lists. It should be noted that a minimal complete definition requires either the definition of *foldr* or *foldMap*, as all other class members are then defined by appropriate defaults. Here is a particular attempt at such defaults:

```
class Foldable t
  where
    fold :: Monoid m => t m -> m
    foldMap :: Monoid m => (a -> m) -> t a -> m
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldl :: (a -> b -> a) -> a -> t b -> a
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl1 :: (a -> a -> a) -> t a -> a
    fold = foldr mappend mempty
    foldMap f = foldr (mappend . f) mempty
    foldr f z = foldr f z . toList
    foldl f z q = foldr (\x g a -> g (f a x)) id q z
    foldr1 f = foldr1 f . toList
    foldl1 f = foldl1 f . toList
```

In a number of places, we leverage a conversion function *toList* for going from a foldable type over an element type to the list type over the same element type. In this manner, we can reduce some operations on foldables to operations on lists. This conversion function is easily defined by a *foldMap* application:

```
toList :: Foldable t => t a -> [a]
toList = foldMap (\x->[x])
```

Looking at the defaults again and their use of *toList*, there is obviously an "unsound" circularity within the definitions, which however would be soundly broken, when either *foldr* or *foldMap* was defined for any given foldable type.

Concept: Higher-kinded polymorphism

Headline

Type parameters of a higher kind than "*"

Illustration

Higher-kinded polymorphism is popular in Language:Haskell with several well-known type classes being parameterized in type constructors rather than types. For instance, the following important Haskell type classes use kind "*->*":

- Type class *Functor*; see the concept of functor.
- Type class *Monad*; see the concept of monad.

For comparison, many popular Haskell type classes are not higher-kinded, i.e., they are parameterized in kind "*", e.g.:

- Type class *Show*.
- Type class *Eq*.

Contribution: haskellTree

Headline

Data processing in Language:Haskell with functors and foldable types

Characteristics

The data structure of a company is converted to a leaf-labeled rose tree which preserves the tree-like shape of the input but otherwise only represents the salary values at the leaves. Thus, names and other properties of departments and employees are not exposed. Such trees are declared as a functor and a foldable type. A bidirectional transformation is then employed to model a salary cut. That is, the company structure is converted to the leaf-labeled tree, then, in turn, to a list, on which to perform salary cut so that finally the modified salaries are integrated back into the company structure.

Illustration

Consider the following sample company:

```
sampleCompany :: Company
sampleCompany =
  Company
    "Acme Corporation"
    [
      Department "Research"
        (Employee "Craig" "Redmond" 123456)
        []
        [
          (Employee "Erik" "Utrecht" 12345),
          (Employee "Ralf" "Koblenz" 1234)
        ],
      Department "Development"
        (Employee "Ray" "Redmond" 234567)
        [
          Department "Dev1"
            (Employee "Klaus" "Boston" 23456)
            [
              Department "Dev1.1"
                (Employee "Karl" "Riga" 2345)
                []
                [(Employee "Joe" "Wifi City" 2344)]
            ]
            []
        ]
        []
    ]
```

When converted to a leaf-labeled rose tree, the sample company looks as follows:

```
sampleTree :: LLTree Float
sampleTree =
  Fork [
    Fork [
      Leaf 123456.0,
      Leaf 12345.0,
      Leaf 1234.0],
    Fork [
      Leaf 234567.0,
      Fork [
        Leaf 23456.0,
        Fork [
          Leaf 2345.0,
          Leaf 2344.0]]]]
```

Here is the corresponding conversion function; it is a get function in the terminology of bidirectional transformation:

```
get :: Company -> LLTree Float
get (Company n ds) = Fork (map getD ds)
```

```
 where
   getD :: Department -> LLTree Float
   getD (Department n m ds es) =  Fork (   [getE m]
                                 ++ map getD ds
                                 ++ map getE es )
     where
       getE :: Employee -> LLTree Float
       getE (Employee     s) = Leaf s
```

Because *LLTree* is a foldable type, it is trivial to further convert the tree to a plain list. Accordingly, salary cut can be expressed at the level of lists. The modified salaries are then put back into the tree with a put function, which we skip here for brevity.

```
cut :: Company -> Company
cut c = put fs' c
 where
   fs = toList (get c)
   fs' = map (/2) fs
```

Architecture

There are these modules:

A data model for Feature:Hierarchical company

```
module Company.Data where

data Company = Company Name [Department]
 deriving (Eq, Read, Show)
data Department = Department Name Manager [Department] [Employee]
 deriving (Eq, Read, Show)
data Employee = Employee Name Address Salary
 deriving (Eq, Read, Show)
type Manager = Employee
type Name = String
type Address = String
type Salary = Float
```

A sample company

```
{- | Sample data of the 101companies System -}

module Company.Sample where

import Company.Data

-- | A sample company useful for basic tests
sampleCompany :: Company
sampleCompany =
  Company
    "Acme Corporation"
    [
      Department "Research"
        (Employee "Craig" "Redmond" 123456)
        []
        [
          (Employee "Erik" "Utrecht" 12345),
          (Employee "Ralf" "Koblenz" 1234)
        ],
      Department "Development"
        (Employee "Ray" "Redmond" 234567)
        [
          Department "Dev1"
            (Employee "Klaus" "Boston" 23456)
            [
              Department "Dev1.1"
                (Employee "Karl" "Riga" 2345)
                []
                [(Employee "Joe" "Wifi City" 2344)]
            ]
            []
        ]
        []
    ]
```

The implementation of Feature:Total

```
module Company.Total where

import Company.Data
import Company.BX
import Data.Foldable
import Data.Monoid

total :: Company -> Float
total = getSum . foldMap Sum . get
```

The implementation of Feature:Cut

```
module Company.Cut where

import Company.Data
import Company.BX
import Data.Foldable
import Data.Monoid

cut :: Company -> Company
cut c = put fs' c
  where
    fs = toList (get c)
    fs' = map (/2) fs
```

A bidirectional transformation

```
module Company.BX where

import Company.Data
import Data.LLTree
import Data.List

get :: Company -> LLTree Float
get (Company n ds) = Fork (map getD ds)
  where
    getD :: Department -> LLTree Float
    getD (Department n m ds es) =  Fork (   [getE m]
                                     ++ map getD ds
                                     ++ map getE es )
      where
        getE :: Employee -> LLTree Float
        getE (Employee _ _ s) = Leaf s

put :: [Float] -> Company -> Company
put fs (Company n ds) = Company n ds'
  where
    ([], ds') = mapAccumL putD fs ds
    putD :: [Float] -> Department -> ([Float], Department)
    putD fs (Department n m ds es) = (fs''', Department n m' ds' es')
      where
        (fs', m') = putE fs m
        (fs'', ds') = mapAccumL putD fs' ds
        (fs''', es') = mapAccumL putE fs'' es
        putE :: [Float] -> Employee -> ([Float], Employee)
        putE (f:fs) (Employee n a s) = (fs, Employee n a f)
```

Leaf-labeled rose trees

```
-- Leaf-labeled rose trees

module Data.LLTree where

import Prelude hiding (foldr, concat)
import Data.Functor
import Data.Foldable

data LLTree a = Leaf a | Fork [LLTree a]
  deriving (Eq, Show, Read)

instance Functor LLTree
  where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Fork ts) = Fork (fmap (fmap f) ts)
```

```
instance Foldable LLTree
  where
    foldr f z (Leaf a) = f a z
    foldr f z (Fork ts) = foldr f z (concat (fmap toList ts))
```

Tests

```
module Main where

import Company.Data
import Company.Sample
import Company.BX
import Company.Total
import Company.Cut
import Data.LLTree
import Data.Foldable (toList)
import Test.HUnit
import System.Exit

sampleTree :: LLTree Float
sampleTree =
  Fork [
    Fork [
      Leaf 123456.0,
      Leaf 12345.0,
      Leaf 1234.0],
    Fork [
      Leaf 234567.0,
      Fork [
        Leaf 23456.0,
        Fork [
          Leaf 2345.0,
          Leaf 2344.0]]]]

sampleTreeList = [123456.0,12345.0,1234.0,234567.0,23456.0,2345.0,2344.0]

totalTest = 399747.0 ~=? total sampleCompany

cutTest = 199873.5 ~=? total (cut sampleCompany)

serializationTest = sampleCompany ~=? read (show sampleCompany)

getTreeTest = sampleTree ~=? get sampleCompany

getTreeListTest = sampleTreeList ~=? toList (get sampleCompany)

tests =
  TestList [
    TestLabel "total" totalTest,
    TestLabel "cut" cutTest,
    TestLabel "serialization" serializationTest,
    TestLabel "getTree" getTreeTest,
    TestLabel "getTreeList" getTreeListTest
  ]

-- | Run all tests and communicate through exit code
main = do
 counts <- runTestTT tests
 if (errors counts > 0 || failures counts > 0)
   then exitFailure
   else exitSuccess
```

The types of

```
module Company.Data where

data Company = Company Name [Department]
 deriving (Eq, Read, Show)
data Department = Department Name Manager [Department] [Employee]
 deriving (Eq, Read, Show)
data Employee = Employee Name Address Salary
 deriving (Eq, Read, Show)
type Manager = Employee
type Name = String
type Address = String
type Salary = Float
```

implement Feature:Closed serialization through Haskell's read/show.

Usage

See https://github.com/101companies/101haskell/blob/master/README.md.

Concept: Bidirectional transformation

Headline

A transformation that can be applied in two directions

Illustration

See Contribution:haskellTree for an in-depth illustration.

Feature: Cut

Headline

Cut the salaries of all employees in half

Description

For a given company, the salaries of all employees are to be cut in half. Let's assume that the management of the company is interested in a salary cut as a response to a financial crisis. Clearly, any real company is likely to respond to a financial crisis in a much less simplistic manner.

Motivation

The feature may be implemented as a transformation, potentially making use of a suitable transformation or data manipulation language. Conceptually, the feature corresponds to a relatively simple and regular kind of transformation, i.e., an iterator-based transformation, which iterates over a company' employees and updates the salaries of the individual employees along the way. It shall be interesting to see how different software languages, technologies, and implementations deal with the conceptual simplicity of the problem at hand.

Illustration

The feature is illustrated with a statement in Language:SQL to be applied to an instance of a straightforward relational schema for companies where we assume that all employees belong to a single company:

```
UPDATE employee
 SET salary = salary / 2;
```

The snippet originates from Contribution:mySqlMany.

Relationships

- See Feature:Total for a query scenario instead of a transformation scenario.
- In fact, Feature:Total is likely to be helpful in a *demonstration* of Feature:Salary cut.
- The present feature should be applicable to any data model of companies, specifically Feature:Flat company and Feature:Hierarchical_company.

Guidelines

- The *name* of an operation for cutting salaries thereof should involve the term "cut". This guideline is met by the above illustration, if we assume that the shown SQL statement is stored in a SQL script with name "Cut.sql". Likewise, if OO programming was used for implementation, then the names of the corresponding methods should involve the term "cut".
- A suitable *demonstration* of the feature's implementation should cut the salaries of a sample company. This guideline is met by the above illustration, if we assume that the shown SQL statement is executed on a database which readily contains company data. Queries according to Feature:Total may be used to compare salaries before and after the cut. All such database preparation, data manipulation, and query execution should preferably be scripted. By contrast, if OO programming was used, then the demonstration could be delivered in the form of unit tests.

Feature: Total

Headline

Sum up the salaries of all employees

Description

The salaries of a company's employees are to be summed up. Let's assume that the management of the company is interested in the salary total as a simple indicator for the amount of money paid to the employees, be it for a press release or otherwise. Clearly, any real company faces other expenses per employee, which are not totaled in this manner.

Motivation

The feature may be implemented as a query, potentially making use of a suitable query language. Conceptually, the feature corresponds to a relatively simple and regular kind of query, i.e., an iterator-based query, which iterates over a company' employees and aggregates the salaries of the individual employees along the way. It shall be interesting to see how different software languages, technologies, and implementations deal with the conceptual simplicity of the problem at hand.

Illustration

Totaling salaries in SQL

Consider the following Language:SQL query which can be applied to an instance of a straightforward relational schema for companies. We assume that all employees belong to a single company; The snippet originates from Contribution:mySqlMany.

```
SELECT SUM(salary) FROM employee;
```

Totaling salaries in Haskell

Consider the following Language:Haskell functions which are applied to a simple representation of companies.

```
-- Total all salaries in a company
total :: Company -> Float
total = sum . salaries

-- Extract all salaries in a company
salaries :: Company -> [Salary]
salaries (n, es) = salariesEs es

-- Extract all salaries of lists of employees
salariesEs :: [Employee] -> [Salary]
salariesEs [] = []
salariesEs (e:es) = getSalary e : salariesEs es

-- Extract the salary from an employee
getSalary :: Employee -> Salary
getSalary ( ,  , s) = s
```

Relationships

- See Feature:Cut for a transformation scenario instead of a query scenario.
- See Feature:Depth for a more advanced query scenario.
- The present feature should be applicable to any data model of companies, specifically Feature:Flat company and Feature:Hierarchical_company.

Guidelines

- The *name* of an operation for summing up salaries thereof should involve the term "total". This guideline is met by the above illustration, if we assume that the shown SQL statement is stored in a SQL script with name "Total.sql". By contrast, if OO programming was used for implementation, then the names of the corresponding methods should involve the term "total".

- A suitable *demonstration* of the feature's implementation should total the salaries of a sample company. This guideline is met by the above illustration, if we assume that the shown SQL statement is executed on a database which readily contains company data. All such database preparation and query execution should preferably be scripted. Likewise, if OO programming was used, then the demonstration could be delivered in the form of unit tests.

Concept: Maybe type

Headline

A polymorphic type for handling optional values and errors

Illustration

In Language:Haskell, maybe types are modeled by the following type constructor:

```
-- The Maybe type constructor
data Maybe a = Nothing | Just a
 deriving (Read, Show, Eq)
```

Nothing represents the lack of a value (or an error). *Just* represent the presence of a value. Functionality may use arbitrary pattern matching to process values of Maybe types, but there is a fold function for maybes:

```
-- A fold function for maybes
maybe :: b -> (a -> b) -> Maybe a -> b
maybe b _ Nothing = b
maybe _ f (Just a) = f a
```

Thus, *maybe* inspects the maybe value passed as the third and final argument and applies the first or the second argument for the cases *Nothing* or *Just*, respectively. Let us illustrate a maybe-like fold by means of looking up entries in a map. Let's say that we maintain a map of abbreviations from which to lookup abbreviations for expansion. We would like to keep a term, as is, if it does not appear in the map. Thus:

```
> let abbreviations = [("FP","Functional programming"),("LP","Logic programming")]
> lookup "FP" abbreviations
Just "Functional programming"
> lookup "OOP" abbreviations
Nothing
> let lookup' x m = maybe x id (lookup x m)
> lookup' "FP" abbreviations
"Functional programming"
> lookup' "OOP" abbreviations
"OOP"
```

Language: Haskell

Headline

The functional programming language Haskell

Details

101wiki hosts plenty of Haskell-based contributions. This is evident from corresponding back-links. More selective sets of Haskell-based contributions are organized in themes: Theme:Haskell data, Theme:Haskell potpourri, and Theme:Haskell genericity. Haskell is also the language of choice for a course supported by 101wiki: Course:Lambdas_in_Koblenz.

Illustration

The following expression takes the first 42 elements of the infinite list of natural numbers:

```
> take 42 [0..]
[0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41]
```

In this example, we leverage Haskell's lazy evaluation.

Concept: Rose tree

Headline

A tree with an arbitrary number of sub-trees per node

Illustration

Such a tree could carry information in all nodes, in which case we speak of a node-labeled rose tree:

```
data NLTree a = NLTree a [NLTree a]
  deriving (Eq, Show, Read)
```

For instance:

```
sampleNLTree =
  NLTree 1 [
    NLTree 2 [],
    NLTree 3 [NLTree 4 []],
    NLTree 5 []]
```

Labeling in a rose tree may also be limited to the leaves, in which case we speak of a leaf-labeled rose tree:

```
data LLTree a = Leaf a | Fork [LLTree a]
  deriving (Eq, Show, Read)
```

For instance:

```
sampleLLTree =
  Fork [
    Leaf 1,
    Fork [Leaf 2],
    Leaf 3]
```

For what it matters, we can make the type constructors for rose trees functors and foldable types:

```
instance Functor NLTree
  where
    fmap f (NLTree x ts) = NLTree (f x) (fmap (fmap f) ts)

instance Foldable NLTree
  where
    foldr f z (NLTree x ts) = foldr f z (x : concat (fmap toList ts))

instance Functor LLTree
  where
    fmap f (Leaf x) = Leaf (f x)
    fmap f (Fork ts) = Fork (fmap (fmap f) ts)

instance Foldable LLTree
  where
    foldr f z (Leaf x) = x `f` z
    foldr f z (Fork ts) = foldr f z (concat (fmap toList ts))
```

The *fmap* definitions basically push *fmap* into the subtrees while using the list instance of *fmap* to process lists of subtrees. The *foldr* definitions basically reduce *foldr* on trees to 'foldr' on lists by apply *toList* on subtrees. Here we note that *toList* can be defined for any foldable type as follows:

```
toList :: Foldable t => t a -> [a]
toList = foldMap (\x->[x])
```

Concept: Applicative functor

Headline

A functor with function application within the functor

Description

Applicative functors are described here briefly in Haskell's sense.

The corresponding type class (modulo some simplifications) looks as follows.

```
class Functor f => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

The expectation is that *pure* promotes a value to a functorial value whereas "" can be seen as a variation of fmap such that a function within the functor (as opposed to just a plain function) is applied to a functorial value.

The following laws are assumed.

```
pure f <*> x = fmap f x
pure id <*> v = v
pure (.) <*> u <*> v <*> w = u <*> (v <*> w)
pure f <*> pure x = pure (f x)
u <*> pure y = pure ($ y) <*> u
```

Illustration

Simple examples

We make *Maybe* and lists applicative functors:

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  (Just f) <*> x = fmap f x

instance Applicative [] where
  pure x = [x]
  fs <*> xs = [ f x | f <- fs, x <- xs ]
```

Thus, in the *Maybe* case, a *Nothing* as a function makes us return a *Nothing* as result, but if the function is available then it is fmapped over the argument. In the list case, we use a list comprehension to apply all available functions too all available values.

The instances can be exercised at the Haskell prompt as follows:

```
> Just odd <*> Just 2
Just False
> [odd, even] <*> [1,2,3,4]
[True,False,True,False,False,True,False,True]
```

To see that applicative functors facilitate function application for functorial values pretty well, consider the following functorial variation on plain function application.

```
(<$>) :: Functor f => (a -> b) -> f a -> f b
f <$> x = fmap f x
```

Consider the following application.

```
> (+) <$> [1,2] <*> [3,4]
[4,5,5,6]
```

Thus, the applicative operator "" is used to line up (any number of) functorial arguments and *fmap* is used for the "rest" of the application.

A more advanced example

We will use now an applicative functor to support environment passing within a recursive computation.

Consider the following interpreter for simple expressions:

```
data Exp
  = Var String
  | Val Int
  | Add Exp Exp

-- Environments with a fetch (lookup) function
type Env = [(String, Int)]
fetch x ((y,v):n) = if x==y then v else fetch x n

-- Straightforward interpreter; we take care of environment passing
eval :: Exp -> Env -> Int
eval (Var x) n = fetch x n
eval (Val v) _ = v
eval (Add e1 e2) n = eval e1 n + eval e2 n
```

We can evaluate expressions like this:

```
> eval (Add (Var "x") (Val 22)) [("x", 20)]
42
```

Let's try to switch to a more combinatorial style such that we abstract from explicit environment passing. To this end, we leverage the so-called SKI combinators:

```
-- More point-free, combinatorial interpreter hiding some environment passing
eval' :: Exp -> Env -> Int
eval' (Var x) = fetch x
eval' (Val v) = k v
eval' (Add e1 e2) = k (+) `s` eval' e1 `s` eval' e2

-- https://en.wikipedia.org/wiki/SKI_combinator_calculus
i :: a -> a
i x = x -- aka id
k :: a -> b -> a
k x y = x -- aka const
s :: (a -> b -> c) -> (a -> b) -> a -> c
s x y z = x z (y z) -- aka <*> of applicative
```

The applicative functor for the instance "(->) a" provides exactly the necessary abstraction:

```
-- Switch to applicative functor style, thereby demonstrating a general pattern
eval'' :: Exp -> Env -> Int
eval'' (Var x) = fetch x
eval'' (Val v) = pure v
eval'' (Add e1 e2) = pure (+) <*> eval'' e1 <*> eval'' e2
```

Contribution: haskellNonfunctorial

Headline

Reusable abstractions for accessing company data

Characteristics

Feature:Total is an example of an operation on company salaries. Other options on company salaries are conceivable, too; see, for example, Feature:Median. Feature:Cut is an example of an operation for transforming companies in the salary position. Other options on company salaries are conceivable, too; see, for example, Feature:Cut. It is quite common to set up reusable abstractions (potentially higher-order functions) to process heterogeneous data structures in all kinds of ways. That is, we generalize over cutting salaries by setting up a transformation function which is parameterized in the function to be applied in salary positions and we generalizing over totaliong salaries by setting up a function to extract all salaries as a list.

Illustration

We total salaries by leveraging the extraction of salaries from companies:

```
total :: Company -> Float
total = sum . getSalariesFromCompany
```

Computing the median salary can rely on the same function for salary extraction:

```
median :: Company -> Float
median c = sort ss!!(length ss `div` 2)
  where
    ss = getSalariesFromCompany c
```

We cut salaries by leveraging the function for transforming salaries in companies:

```
cut :: Company -> Company
cut = transformSalariesInCompany (/2)
```

Raising salaries can rely on the same function for transforming salaries in companies:

```
raise :: Company -> Company
raise = transformSalariesInCompany (*1.01)
```

Architecture

Salary extraction

```
getSalariesFromCompany :: Company -> [Salary]
getSalariesFromCompany (Company n ds) = concat ds'
  where
    ds' = map fromD ds
    fromD (Department n m ds es) = m' : concat ds' ++ es'
      where
        m' = fromE m
        ds' = map fromD ds
        es' = map fromE es
        fromE (Employee _ _ s) = s
```

Salary transformation

```
transformSalariesInCompany :: (Salary -> Salary) -> Company -> Company
transformSalariesInCompany f (Company n ds) = Company n ds'
  where
    ds' = map inD ds
    inD (Department n m ds es) = Department n m' ds' es'
      where
        m' = inE m
        ds' = map inD ds
        es' = map inE es
        inE (Employee n a s) = Employee n a (f s)
```

Discussion

Heterogeneous data structures such as companies breaking down into departments, employees, names, addresses, and salaries. We could set up transoformation and extraction helpers for other ingredients of compoanies. See Contribution:haskellFunctorial for an alternative approach of organizing access to positions of a certain type. Ultimately, we could leverage Theme:Haskell genericity to perform traversals on heterogeneous data structures; see, for example, Contribution:haskellSyb.

Contribution: haskellFunctorial

Headline

Using functorial map and fold to access company data

Characteristics

Company data is quite hetereogenous. It's a container of kinds. There is a company (at the top); there is departments and employees in the tree-like structure of a company; there is also names (of employees, departments, and companies); further, there are addresses (of employess), and there are salaries of employees including managers as a special kind of employees.

We would like to access company structures in a functorial style. We have in mind the common operations Feature:Total and Feature:Cut. For these operations to fit into the functorial framework, we need to parametrize the company types appropriately in terms of salary-type positions.

Illustration

We need companies and descendants to be parametrized in salaries:

```
data Company s = Company Name [Department s]
...
```

We perform salary cut by functioral map:

```
cut :: Company Float -> Company Float
cut = fmap (/2)
```

To this end, we assume the `Company` type to be parametrized in salary positions.

Likewise, we total salaries by an application of the foldr function:

```
total :: Company Float -> Float
total = foldr (+) 0
```

Here, we leverage the fact that we can access all type-parameter positions in a monoidal manner and hence essentially reduce all salaries.

Architecture

Company data model parameterized in salary position

We set up compamy data in a parameterized manner as follows:

```
-- The data model is parameterized in what's going to be Float-based salaries
data Company s = Company Name [Department s]
 deriving (Eq, Read, Show)
data Department s = Department Name (Manager s) [Department s] [Employee s]
 deriving (Eq, Read, Show)
data Employee s = Employee Name Address s
 deriving (Eq, Read, Show)
type Manager s = Employee s
type Name = String
type Address = String
```

Companies as functors over salaries

Here are the `Functor` instances:

```
instance Functor Company
  where
    fmap f (Company n ds) = Company n ds'
      where
        ds' = map (fmap f) ds

instance Functor Department
```

```
  where
    fmap f (Department n m ds es) = Department n m' ds' es'
      where
        m' = fmap f m
        ds' = map (fmap f) ds
        es' = map (fmap f) es

instance Functor Employee
  where
    fmap f (Employee n a s) = Employee n a (f s)
```

Companies as foldables over salaries

Here are also the `Foldable` instances -- we exploit the property of the `Foldable` type class, that the minimal definition in an instance can consist of either `foldr` or `foldMap`, as the counterpart and all other members of `Foldable` are universally predefined; it turns out that `foldMap` is straightforward to define for companies, even though one could argue that the following code isn't efficient, for example, in terms of the use of `concat`.

```
instance Foldable Company
  where
    foldMap f (Company _ ds) = ds'
      where
        ds' = mconcat (map (foldMap f) ds)

instance Foldable Department
  where
    foldMap f (Department _ m ds es) = m' `mappend` ds' `mappend` es'
      where
        m' = foldMap f m
        ds' = mconcat (map (foldMap f) ds)
        es' = mconcat (map (foldMap f) es)

instance Foldable Employee
  where
    foldMap f (Employee _ _ s) = f s
```

Discussion

The example demonstrates an important limitation of the functorial approach: we need to assume that a data structure can be usefully parameterized in an "element" type of a "container" type. This makes sesme for actual container types such as lists or rose trees, but it is not entirely useful for more heterogeneous data structures such as companies breaking down into departments, employees, names, addresses, and salaries. In theory, we could expose "views" on copanies so that the substructure of interested is "exposed" via the type parameter, but such a conversion back and forth between custom views would be both expensive and possibly confusing.

See Contribution:haskellNonfunctorial for a more idiomatoc approach of generalizing cut to a higher-order function. Ultimately, we could leverage Theme:Haskell genericity to perform traversals on heterogeneous data structures; see, for example, Contribution:haskellSyb.
