Script: Type-class polymorphism

Headline

Lecture "Type-class polymorphism" as part of Course:Lambdas in Koblenz

Description

We have looked at parametric polymorphism as a means to describe functionality in a universal way for many types or, in fact, all types of a certain kind. This is appropriate whenever such polymorphic functionality does not need to make any assumptions about the actual types that fill in the type parameters eventually. There is another kind of polymorphism, where the same kind of functionality (in terms of function signatures) needs to be defined for many types, but these definitions may vary per type. For instance, the conversion of values to text is required functionality for many types, but its definition depends on the input type. Type class support such polymorphism to which we may refer thus as type-class polymorphism or bounded polymorphism (or "overloading"). Other use cases for type-class polymorphism are equality, total ordering, number types, algebraic structures such as monoids, and traversal of data (containers).

Concepts

- Polymorphism
  - Parametric polymorphism
  - Bounded polymorphism
- Type class
  - Type-class instance
  - Type-class constraint
  - Type-class polymorphism
- Equality
  - Structural equality
  - Semantic equality
- Total order
- Monoid
  - List monoid
  - Sum monoid
  - Product monoid
- Foldable type
  - List type
  - Maybe type
  - Rose tree

Languages

- Language:Haskell

Features

These features are eligible to the use of monoids for implementation.

- Feature:Total
- Feature:Depth
- Feature:Ranking
- Feature:Mentoring

Contributions

- Contribution:haskellProfessional: Feature implementations without the use of monoids.
- Contribution:haskellMonoid: Feature implementations with the use of monoids.

Metadata

- Course:Lambdas in Koblenz
- Script:Higher-order_functions_in_Haskell

Concept: Sum monoid

Headline

A monoid leveraging addition for the associative operation

Illustration

Number types may be completed into monoids in different ways. The sum monoid favors addition for the associative operation of the monoid. We illustrate the sum monoid in Language:Haskell on the grounds of the type class *Monoid* with the first two members needed for a minimal complete definition:

```
-- The type class Monoid
class Monoid a
  where
    mempty :: a -- neutral element
    mappend :: a -> a -> a -- associative operation
    mconcat :: [a] -> a -- fold
    mconcat = foldr mappend mempty
```

The sum monoid relies on a designated type which essentially wraps a number type:

```
-- The type of the sum monoid
newtype Sum a = Sum { getSum :: a }
```

Here is the type-class instance for the sum monoid:

```
-- The Monoid instance for numbers under addition
instance Num a => Monoid (Sum a)
  where
    mempty = Sum 0
    x `mappend` y = Sum (getSum x + getSum y)
```

For further illustration, we can reconstruct the standard *sum* function in a monoidal way. To this end, we first review the normal definition in terms of foldr:

```
-- A foldr-based definition of sum
sum' :: Num a => [a] -> a
sum' = foldr (+) 0

-- A monoidal definition of sum
sum'' :: Num a => [a] -> a
sum'' = getSum . mconcat . map Sum
```

Metadata

- Monoid
- Product monoid

Concept: Type-class constraint

Headline

Constraint on the type parameter of a type class or an instance

Illustration

Type-class constraints define bounds on type parameters used in the declaration of type classes or type-class instances (Ã la Language:Haskell). In this manner, type-class constraints feed into another form of bounded polymorphism.

Consider, for example, the following type-class instance for equality of pairs:

```
-- Equality of pairs
instance (Eq a, Eq b) => Eq (a,b)
  where
    x == y = fst x == fst y && snd x == snd y
```

Clearly, such equality needs to be defined in a component-wise manner: for the first ("fst") and the second ("snd") project of a pair. In the interest of polymorphism, the type of the components should not be fixed, but the availability of equality needs to be assumed for the component types. Thus, the two constraints in the head of the instance:

```
instance (Eq a, Eq b) => ...
```

Likewise, a type class may also be constrained. Consider, for example, the following type class for comparison:

```
class Eq a => Ord a where
  compare :: a -> a -> Ordering
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (<=) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
```

Please observe the constraint:

```
class Eq a => ...
```

This class contains a constraint such that total order (comparison) can only be defined for types with equality. This is effectively a sanity check for the programmer because comparison subsumes the case for equality, conceptually. Without the constraint, a programmer may accidentally forget to implement equality, explicitly.

Importantly, type-class constraints propagate through (inferred) types of expressions. Consider these illustrations of inferring types of expressions at the interpreter prompt:

```
> :t (==)
(==) :: Eq a => a -> a -> Bool
> :t (==42)
(==42) :: (Eq a, Num a) => a -> Bool
```

Metadata

- Concept

Concept: List type

Headline

A data type of lists for some element type

Metadata

- http://en.wikipedia.org/wiki/List
- Data type
- Vocabulary:Data structure

Concept: Bounded polymorphism

Headline

A form of polymorphism with a bound for feasible actual type parameters

Illustration

- See polymorphism for a simple illustration.
- See type classes for a more profound illustration.

Metadata

- Polymorphism
- http://en.wikipedia.org/wiki/Bounded_quantification
- http://en.wikipedia.org/wiki/Polymorphism_(computer_science)

Course: Lambdas in Koblenz

Headline

Introduction to functional programming at the University of Koblenz-Landau

Schedule

- Lecture First steps
- Lecture Basic software engineering
- Lecture Searching and sorting
- Lecture Data modeling in Haskell
- Lecture Functional data structures
- Lecture Higher-order functions
- Lecture Type-class polymorphism
- Lecture Functors and friends
- Lecture Unparsing and parsing
- Lecture Monads
- Lecture Generic functions

Metadata

- http://softlang.wikidot.com/course:fp

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 ofMaybe type for another simple example of a foldable type. See the concept ofrose 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.

Metadata

- http://www.haskell.org/haskellwiki/Foldable_and_Traversable
- Vocabulary:Functional programming
- Concept

Contribution: haskellProfessional

Headline

Idiomatic implementation of several feature in Language:Haskell..

Characteristics

The objective of this contribution is to show idiomatic Haskell code for many functional and data requirements. We leave out features which would require extra programming technologies as those would be covered by designated contributions. Also, some data requirements are left out as they deal with more specialized features of the system:Company. There are several other Language:Haskell-based contributions (specifically those in Theme:Haskell introduction) that address smaller feature sets and limit their use of language features or focus on specific idioms (for pedagogical reasons). So the present contribution is more of an attempt as to how a knowledgeable Haskell programmer would possibly approach the features in serious way.

Relationships

- The present contribution is engineered in much the same way as Contribution:haskellEngineer.
- The present contribution uses the same data model as Contribution:haskellComposition, which is also reused by yet other contributions.

Usage

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

Metadata

- Feature:Hierarchical company
- Feature:Total
- Feature:Median
- Feature:Cut
- Feature:Depth
- Feature:Mentoring
- Feature:Ranking
- Feature:Closed serialization
- Language:Haskell
- Language:Haskell 98
- Technology:GHC
- Technology:Cabal
- Technology:HUnit
- Technology:Haddock

- Contributor:rlaemmel
- Theme:Haskell introduction
- Contribution:haskellEngineer
- Contribution:haskellComposition

Concept: Polymorphism

Headline

The ability of program fragments to operate on elements of several types

Illustration

Consider the type of list append inLanguage:Haskell:

```
(++) :: [a] -> [a] -> [a]
```

This type signature uses a type variable *a* to express that list append is polymorphic in the element type *a*. The operation can be applied for as long as the element type of both operand lists for an append are the same. We also speak of parametric polymorphism in this case.

Consider the type of addition inLanguage:Haskell:

```
(+) :: Num a => a -> a -> a
```

This type signature uses a type constraint on the operand type of addition to express that only "suitable" types (i.e., type-class instances of *Num*) can be used for addition. We also speak of type-class polymorphism or more generally ofbounded polymorphism in this case. Languages with subtyping may also use types in a subtyping hierarchy for bounds.

Metadata

- http://en.wikipedia.org/wiki/Polymorphism (computer science)
- http://en.wikipedia.org/wiki/Polymorphism in object-oriented programming
- Vocabulary:Programming language
- Concept

Contribution: haskellMonoid

Headline

Modeling queries in Language:Haskell with the help of monoids

Characteristics

Several functional requirements are implemented while making explicit use the monoids. For instance, Feature:Total is implemented with the help of the sum monoid. Only those functional requirements are implemented that indeed may benefit from monoids as such. For instance, Feature:Cut is not implemented.

Illustration

Consider the implementation of Feature:Total:

```
-- | Total all salaries in a company
total :: Company -> Float
total (n, ds) = getSum (mconcat (map totalD ds))
  where
    -- Total all salaries in a department
    totalD :: Department -> Sum Float
    totalD (Department _ m ds es)
      = mconcat (totalE m : map totalD ds ++ map totalE es)
      where
        -- Extract the salary from an employee
        totalE :: Employee -> Sum Float
        totalE (_, _, s) = Sum s
```

That is, lists of departments and employees are processed by the map function resulting in lists of intermediate results in the monoid's *Sum* type to be reduced accordingly by the monoid's *mconcat* operation, which, in turn, is uniformly defined by applying the fold function to the monoid's binary operation and its identity.

Relationships

- The data model of Contribution:haskellComposition is reused.
- See Contribution:haskellProfessional for an implementation of all relevant features without the use of monoids.

Architecture

There are these modules:

```
{-| A data model for the 101companies System -}
```

```
module Company.Data where

-- | A company consists of name and top-level departments
type Company = (Name, [Department])

-- | A department consists of name, manager, sub-departments, and employees
data Department = Department Name Manager [Department] [Employee]
 deriving (Eq, Read, Show)

-- | An employee consists of name, address, and salary
type Employee = (Name, Address, Salary)

-- | Managers as employees
type Manager = Employee

-- | Names of companies, departments, and employees
type Name = String

-- | Addresses as strings
type Address = String

-- | Salaries as floats
type Salary = Float
```

: a data model for Feature:Hierarchical company

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

module Company.Sample where

import Company.Data

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

: a sample company

```
{-| The operation of totaling all salaries of all employees in a company -}

module Company.Total where

import Company.Data
import Data.Monoid

-- | Total all salaries in a company
total :: Company -> Float
total (n, ds) = getSum (mconcat (map totalD ds))
  where
    -- Total all salaries in a department
    totalD :: Department -> Sum Float
    totalD (Department _ m ds es)
      = mconcat (totalE m : map totalD ds ++ map totalE es)
      where
        -- Extract the salary from an employee
        totalE :: Employee -> Sum Float
        totalE (_, _, s) = Sum s
```

: the implementation of Feature:Total

```
{-| The operation to compute the nesting depth of departments in a company -}

module Company.Depth where

import Company.Data
import Data.Monoid
import Data.Max

-- | Compute the nesting depth of a company
depth :: Company -> Int
depth (_, ds) = max' (map depth' ds)
  where
    -- Maximum of a list of natural numbers
    max' = maybe 0 id . getMax . mconcat
    -- Helper at the department level
    depth' :: Department -> Max Int
    depth' (Department _ _ ds _) = setMax (1 + max' (map depth' ds))
```

: the implementation of Feature:Depth

```
{-| The constraint to check that salaries follow ranks in company hierarchy -}

module Company.Ranking where

import Company.Data
import Data.Monoid
import Data.Max

-- | Check that salaries follow ranks in company hierarchy
ranking :: Company -> Bool
ranking (_, ds) = and (map ranking' ds)
  where
    -- Helper at the department level
    ranking' :: Department -> Bool
    ranking' (Department _ m ds es)
        =  and (map ranking' ds)
```

```
    && maybe True (<getSalary m) (getMax subunits)
   where
    -- Maximum of salaries for immediate employees
    employees :: Max Float
    employees = mconcat (map (setMax . getSalary) es)
    -- Maximum of salaries for immediate sub-departments' managers
    managers :: Max Float
    managers = mconcat (map (setMax . getManagerSalary) ds)
    -- "employees" and "managers" combined
    subunits :: Max Float
    subunits = managers `mappend` employees
  -- Extract the salary of a department's manager
  getManagerSalary :: Department -> Float
  getManagerSalary (Department _ m _ _) = getSalary m
  -- Extract the salary of an employee
  getSalary :: Employee -> Float
  getSalary (_, _, s) = s

-- | A company that violates the ranking constraint
rankingFailSample =
 ( "Fail Industries",
  [ Department "Failure"
    ("Ubermanager", "Top Floor", 100)
    []
    [("Joe Programmer", "Basement", 1000)]
  ]
 )
```

: the implementation of Feature:Ranking

```
{-| A monoid for optional maxima -}

module Data.Max (
 Max,
 getMax,
 setMax,
 noMax
) where

import Data.Monoid

-- | A data type for maxima without default
data Ord x =>
   Max x = Max {
    -- | Retrieve maximum, if any
    getMax :: Maybe x
   }

-- | Set max to "just" a value
setMax :: Ord x => x -> Max x
setMax = Max . Just

-- | The absent maximum
noMax :: Ord x => Max x
noMax = Max { getMax = Nothing }

-- | A monoid for maxima
instance Ord x => Monoid (Max x)
 where
  mempty = Max Nothing
```

```
   x `mappend` y
     = case (getMax x, getMax y) of
         (Nothing, m) -> Max m
         (m, Nothing) -> Max m
         (Just m1, Just m2) -> Max (Just (m1 `max` m2))
```

: a monoid for optional maxima

```
{-| Tests for the 101companies System -}

module Main where

import Company.Data
import Company.Sample
import Company.Total
import Company.Depth
import Company.Ranking
import Test.HUnit
import System.Exit

-- | Compare salary total of sample company with baseline
totalTest = 399747.0 ~=? total sampleCompany

-- | Compare depth of sample company with baseline
depthTest = 3 ~=? depth sampleCompany

-- | Check ranking constraint for salaries of sample company
rankingOkTest =  True ~=? ranking sampleCompany

-- | Negative test case for ranking constraint
rankingFailTest = False ~=? ranking rankingFailSample

-- | Test for round-tripping of de-/serialization of sample company
serializationTest = sampleCompany ~=? read (show sampleCompany)

-- | The list of tests
tests =
  TestList [
    TestLabel "total" totalTest,
    TestLabel "depth" depthTest,
    TestLabel "rankingOk" rankingOkTest,
    TestLabel "rankingFail" rankingFailTest,
    TestLabel "serialization" serializationTest
  ]

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

: Tests The types of

```
{-| A data model for the 101companies System -}

module Company.Data where

-- | A company consists of name and top-level departments
type Company = (Name, [Department])
```

```
-- | A department consists of name, manager, sub-departments, and employees
data Department = Department Name Manager [Department] [Employee]
 deriving (Eq, Read, Show)

-- | An employee consists of name, address, and salary
type Employee = (Name, Address, Salary)

-- | Managers as employees
type Manager = Employee

-- | Names of companies, departments, and employees
type Name = String

-- | Addresses as strings
type Address = String

-- | Salaries as floats
type Salary = Float
```

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

Usage

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

Metadata

- Language:Haskell
- Language:Haskell 98
- Technology:GHC
- Technology:Cabal
- Technology:HUnit
- Feature:Hierarchical company
- Feature:Total
- Feature:Depth
- Feature:Ranking
- Feature:Closed serialization
- Contributor:rlaemmel
- Theme:Haskell introduction
- Contribution:haskellProfessional
- Contribution:haskellProfessional
- Contribution:haskellProfessional

Concept: Semantic equality

Headline

Equality with taking into account semantics of data

Illustration

We speak of semantic equality when we take semantic properties of the underlying data into account. Semantic equality is to be contrasted with structural equality.

Consider the following type for the representation of arithmetic expressions:

```
-- Simple arithmetic expressions
data Expr = Const Int | Add Expr Expr
```

When assuming straightforward structural equality, then the following properties should hold:

```
> Const 42 == Const 42
True
> Const 42 == Add (Const 20) (Const 22)
False
```

The second equality test fails because the constant term is clearly structurally unequal to the addition term. Let us take semantic properties of the underlying data into account. One option for the given example is that we say that two arithmetic expressions are equal if and only if they evaluate to the same result. In Haskell, this is expressed with the following type-class instance for the type class *Eq*:

```
-- Equality based on evaluation
instance Eq Expr
  where
    x == y = eval x == eval y
      where
        eval (Const i) = i
        eval (Add e1 e2) = eval e1 + eval e2
```

For instance:

```
*Main> Const 42 == Add (Const 20) (Const 22)
True
*Main> Const 42 == Const 41
False
```

Semantic equality based on proper evaluation does not quite generalize because, we may not be able to evaluate the structure at hand. Think of arithmetic expressions, for example, when they contain free variables. Often, semantic equality is defined relative to selected semantic properties that are readily attainable. For instance, consider semantic equality of our arithmetic expressions modulo associativity of addition. In Haskell, this is expressed as follows:

```
-- Lawful equality
instance Eq Expr
  where
    x == y = eq (normalize x) (normalize y)
      where

        -- Associate addition to the right
        normalize :: Expr -> Expr
        normalize x@(Const i) = x
        normalize (Add x@(Const i) y) = Add x (normalize y)
        normalize (Add (Add x y) z) = normalize (Add x (Add y z))

        -- Uniform (structural) equality
        eq :: Expr -> Expr -> Bool
        eq (Const i) (Const j) = i == j
        eq (Add e1 e2) (Add e3 e4) = eq e1 e3 && eq e2 e4
        eq _ _ = False
```

For instance:

```
> let c1 = Const 1
> let c2 = Const 2
> let c3 = Const 3
> Add c1 (Add c2 c3) == Add (Add c1 c2) c3
True
```

Metadata

- Equality
- Structural equality

Feature: Mentoring

Headline

Associate employees in terms of mentoring

Description

Employees may sign up for a mentor. The idea is that mentors help their mentees generally with career management. Operationally, a mentee may consult his or her mentor, for example, to interpret results of a performance appraisal and to draw appropriate conclusions. As far as the system:Company is concerned, it suffices to merely maintain mentors so that management knows about everything.

The association for mentorship is constrained as follows:

- Each employee may have one associated *mentor*.
- Each employee may have any number of associated *mentees*.
- Mentors and mentees are employees (managers or not).
- *A* is mentor of *B* iff *B* is mentee of *A*.
- An employee cannot be a mentor of him- or herself.

Arguably, further constraints could be added. (For instance, it may be reasonable to require that if *A* is mentor of *B*, then *B* must not be mentor of *A*. In this manner, direct cycles would be forbidden.)

Bidirectional navigation is required for the mentorship association.

Motivation

The feature is interesting in so far that it requires more general associations and graph shape as opposed to just composition and tree shape for the basic hierarchical organization of companies according to Feature:Hierarchical company. That is, while companies and departments are decomposed in a tree-like manner, mentorship links may reach across the organizational structure. Further, bidirectional navigation as opposed to the simpler unidirectional navigation is required. In a Language:UML class diagram, for example, the mentorship association can be modeled in a straightforward way. In a functional programming language and pure style, the association's implementation necessitates look-up functions for locating linked employees, possibly identified by name. In an OO programming language with references, the mere links for mentorship are implemented easily, but bidirectional navigation and the above constraints necessitates encoding, unless first-class relationships were available in the programming language.

Illustration

The feature is illustrated with predicates in Language:Datalog. That is, there are declarations of predicates mentorOf/2 and menteeOf/2 to relate employees in both navigation directions of the association. The clauses implement the above description; see the comments for clarification.

```
// Each employee may have a mentor (in the same company or not).
mentorOf[tee] = tor -> Employee(tee), Employee(tor).

// Each employee may have several mentees.
menteeOf(tor,tee) -> Employee(tor), Employee(tee).

// mentorOf and menteeOf are compatible one way.
mentorOf[tee] = tor -> menteeOf(tor,tee).

// mentorOf and menteeOf are compatible the other way.
menteeOf(tor,tee) -> mentorOf[tee] = tor.

// In fact, menteeOf is derived from mentorOf.
menteeOf(tor,tee) <- mentorOf[tee] = tor.

// One must not mentor her- or himself.
mentorOf[tee] = tor -> ! tor = tee.
```

The snippet originates from Contribution:heavyLb.

Relationships

- The present feature should be applicable to any data model of companies, specifically Feature:Flat company and Feature:Hierarchical company.

Guidelines

- Bidirectional navigation is required for the mentorship association.
- The *name* for the direction from mentees to mentors should involve the term "mentor" (e.g., "getMentor"). The *name* for the opposite direction should involve the term "mentee" or "mentees" (e.g., "getMentees").
- A suitable *demonstration* of the feature's implementation should link some employees according to the association and navigate the association in both directions for some employees.

Metadata

- http://en.wikipedia.org/wiki/Mentorship
- http://en.wikipedia.org/wiki/Performance appraisal
- Data requirement
- Optional feature
- Graph

Feature: Depth

Headline

Compute the nesting depth of departments

Description

The nesting depth of departments within a company is to be computed; see below for details. Let's assume that the management of the company is interested in the nesting depth as a simple indicator for the complexity of the company (or particular departments thereof) in the sense of a hierarchical organization. Nesting depth, together possibly with other metrics and information, could feed into the discussion of reorganizing business structures.

The nesting depth is computed as follows:

- The depth of a department is 1 + the maximum of the depths of its sub-departments.
- In particular, the depth of a department without sub-departments is 1.
- The depth of a company is the maximum of the depths of its (immediate) departments.

Motivation

The feature may be implemented as a query, potentially making use of a suitable query language. Conceptually, the required query is non-trivial in that it needs to process company structure recursively so that nesting of departments can be properly observed. For instance, it is not straightforward to design a Language:SQL query that computes indeed nesting depth on a normalized relational schema for company data. Thus, it shall be interesting to see how different software languages, technologies, and implementations succeed in realizing the feature.

Illustration

The feature is illustrated with a Function in Language:Haskell that works on top of appropriate algebraic data types for company data; the function recurses into company data in a straightforward manner and it counts departments along the way:

```
depth :: Company -> Int
depth (Company   ds) = max' (map depth' ds)
  where
    max' = foldr max 0
    depth' :: Department -> Int
    depth' (Department     ds  ) = 1 + max' (map depth' ds)
```

The snippet originates from Contribution:haskellComposition.

Relationships

- See Feature:Total for a simpler query scenario.
- Indeed, the present feature should be tackled only afterFeature:Total.
- The present feature can only usefully instantiated on top ofFeature:Hierarchical_company, as it assumes nesting of departments for non-trivial results.

Guidelines

- The *name* of an operation for computing the nesting depth of departments should involve the term "depth".
- A suitable *demonstration* of the feature's implementation should compute the depth of a sample company.
- See Feature:Total for more detailed guidelines on a query scenario, which apply similarly to the present feature.

Metadata

- http://en.wikipedia.org/wiki/Hierarchical organization
- http://en.wikipedia.org/wiki/Restructuring
- Functional requirement
- Query
- Optional feature
- Feature:Hierarchical_company

Feature: Ranking

Headline

Check salaries to follow ranks in company hierarchy

Description

Any company needs a pay structure (say, pay model). The present feature describes a constraint for a particular pay structure. Within each department, the salary of the department's manager is higher than all salaries of a department's immediate and sub-immediate employees. The constraint needs to be checked or enforced along construction and the modification of company data. Clearly, this is just one possible and arguably rather rigid and unrealistic pay structure.

Motivation

Conceptually, the feature imposes a global invariant on company data. Straightforward expressiveness of type systems is not sufficient to model the constraint. Simple contracts in the sense of pre- and post-conditions or class invariants on local state are also not sufficient; we need to allow for traversal of object graphs. Of course, the constraint can be expressed more or less easily as a recursive computation, very much like a query over the hierarchical structure of companies; see Feature:Depth.

Illustration

The feature is illustrated with a Function in Language:Haskell that works on top of appropriate algebraic data types for company data; the function recurses into company data in a straightforward manner and it counts departments along the way:

```
align :: Company -> Bool
align (Company   ds) = and (map (align' Nothing) ds)
 where
  align' :: Maybe Float -> Department -> Bool
  align' v (Department   m ds es)
    =  maybe True (>getSalary m) v
    && and (map (align' (Just (getSalary m))) ds)
    && and (map ((<getSalary m) . getSalary) es)
  getSalary :: Employee -> Float
  getSalary (Employee     s) = s
```

Further, in some code locations the constraint needs to be invoked. Here is some snippet that shows how the constraint is invoked past cutting salaries:

```
main = do
    ... -- code omitted
```

```
-- Cut all salaries
let company' = cut company

-- Test that salaries align with hierarchy
if not (align company')
  then error "constraint violated"
  else return ()
```

The snippet originates from Contribution:haskellComposition.

Relationships

- The present feature can only usefully instantiated on top ofFeature:Hierarchical_company, as it assumes nesting of departments for non-trivial results.
- A straightforward scenario for testing the present feature would check the constraint past cutting salaries according to Feature:Cut.

Guidelines

- The *name* of a constrain for checking alignment of salaries with hierarchical company structure should involve the term "align".
- A suitable *demonstration* of the feature's implementation should show the constraint is to be invoked (explicitly or implicitly) past construction or modification of company data.

Metadata

- http://www.aafp.org/fpm/2000/0200/p30.html
- http://www.ehow.com/info 12076331 alternative-pay-structures-salaried-employees.html
- http://www.slideshare.net/aaronphamilton/strategic-compensation-structure-egalitarian-v-hierarchical
- http://papers.ssrn.com/sol3/papers.cfm?abstract id=74303
- http://en.wikipedia.org/wiki/Hierarchical organization
- Data requirement
- Optional feature
- Feature:Hierarchical_company

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.

Metadata

- Optional feature
- Functional requirement
- Aggregation

Concept: Equality

Headline

Some kind of equality in programming

Illustration

Let us focus here on equality of data as it is used in programming. Different kinds of equality exist: structural equality, semantic equality, reference equality, and possibly others. For example, trivially, the following equalities or inequalities hold as demonstrated at the interpreter prompt of Language:Haskell:

```
> 42 == 42
True
> 42 == 41
False
> True == True
True
> "Foo" == "Bar"
False
```

In various programming languages, equality may be defined by the programmer. For instance, Language:Haskell designates a type class *Eq* to equality (readily defined in the Haskell Prelude:

```
-- A type class for equality
class Eq a
  where
    (==) :: a -> a -> Bool
```

For instance, equality of Booleans would be defined by the following type-class instance:

```
-- Equality of Booleans
instance Eq Bool
  where
    True == True = True
    False == False = True
    _ == _ = False
```

More interestingly, equality of lists would be defined such that the two lists need to be of the same length and their elements need to be equal in a pairwise manner; thus we also need equality for the element type, which is expressed by the extra constraint in the instance:

```
-- Equality of lists
instance Eq a => Eq [a]
  where
    x == y =  length x == length y
           && and (map (uncurry (==)) (zip x y))
```

Metadata

- http://en.wikipedia.org/wiki/Equality_(mathematics)
- http://en.wikipedia.org/wiki/Inequality_(mathematics)
- http://en.wikipedia.org/wiki/Relational_operator#Equality
- Concept

Concept: List monoid

Headline

A monoid for appending lists

Illustration

We illustrate the list monoid in Language:Haskell on the grounds of the type class *Monoid* with the first two members needed for a minimal complete definition:

```
-- The type class Monoid
class Monoid a
  where
    mempty :: a -- neutral element
    mappend :: a -> a -> a -- associative operation
    mconcat :: [a] -> a -- fold
    mconcat = foldr mappend mempty
```

Lists form a monoid in the following way:

```
-- The Monoid instance for lists
instance Monoid [a]
  where
    mempty = []
    mappend = (++)
    mconcat = concat
```

Now it is interesting to observe how *concat* is (or could be) defined:

```
-- Appending many lists
concat :: [[a]] -> [a]
concat = foldr (++) []
```

Please observe the above default definition of *mconcat* within the type class *Monoid*; it generalizes this sort of fold and thus, the definition of *mconcat* would not be needed in the case of the list instance of *Monoid*.

Metadata

- Monoid

Concept: Maybe type

Headline

A polymorphic type for handling optional values and errors

Illustration

In Language:Haskell, maybe types are modeled by the followingtype 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"
```

Metadata

- Vocabulary:Haskell
- http://www.haskell.org/haskellwiki/Maybe

Concept: Monoid

Headline

A type with an associative operation and a neutral element

Illustration

The notion of monoid is precisely defined in group theory, but we focus here on its illustration in a programming setting. Specifically, in functional programming, a monoid is essentially a type with an associative operation and a neutral element. For instance, lists form a monoid with the empty list as neutral element and list append as the associative operation. Monoids are useful, for example, in aggregating results.

In Language:Haskell, monoids are modeled through thetype class *Monoid* with first two members needed for a minimal complete definition:

```
-- The type class Monoid
class Monoid a
  where
    mempty :: a -- neutral element
    mappend :: a -> a -> a -- associative operation
    mconcat :: [a] -> a -- fold
    mconcat = foldr mappend mempty
```

Algebraically, the following properties are required for any monoid (given in Haskell notation):

```
mempty `mappend` x = x -- left unit
x `mappend` mempty = x -- right unit
x `mappend` (y `mappend` z) = (x `mappend` y) `mappend` z -- associativity
```

See the following monoids for continued illustration:

- List monoid
- Sum monoid
- Product monoid

Metadata

- Data type
- Vocabulary:Functional programming
- Vocabulary:Mathematics
- http://en.wikipedia.org/wiki/Monoid
- http://mathworld.wolfram.com/Monoid.html
- http://en.wikibooks.org/wiki/Haskell/Monoids
- http://www.haskell.org/ghc/docs/latest/html/libraries/base/Data-Monoid.html

Concept: Parametric polymorphism

Headline

A form of polymorphism applying to all types of a certain kind

Illustration

See polymorphism.

Metadata

- Polymorphism
- http://en.wikipedia.org/wiki/Parametric_polymorphism
- http://en.wikipedia.org/wiki/Polymorphism_(computer_science)

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'slazy evaluation.

Metadata

- http://www.haskell.org/
- http://en.wikipedia.org/wiki/Haskell_(programming_language)
- Functional programming language

Concept: Structural equality

Headline

Equality in terms of structure alone, without interpretation

Illustration

Structural equality means that two expressions are equal if and only if they agree on constructors or primitive values at every level and in every position. In Haskell, this would be captured by the following type-class instance for the type class *Eq*:

```
-- Simple arithmetic expressions
data Expr = Const Int | Add Expr Expr

-- Uniform (structural) equality
instance Eq Expr
  where
    (Const i) == (Const j) = i == j
    (Add e1 e2) == (Add e3 e4) = e1 == e3 && e2 == e4
    _ == _ = False
```

Thus, the operands of equality (i.e., "==") must agree on the outermost constructor and equality must hold recursively for all immediate positions. For instance:

```
> Const 42 == Const 42
True
> Const 42 == Add (Const 20) (Const 22)
False
```

The second equality test fails because the constant term is clearly structurally unequal to the addition term, even though we can see that both expressions would evaluate to the same result. Indeed, sometimes, we could prefer semantic equality; this is when we take semantic properties of the underlying data into account.

Metadata

- Equality
- Semantic equality

Concept: Product monoid

Headline

A monoid leveraging multiplication for the associative operation

Illustration

Number types may be completed into monoids in different ways. Most notably, either addition or multiplication can be used for the associative operation of the monoid. See the concept of the sum monoid for a detailed illustration when addition is favored. Obviously, all given definitions would be routinely adapted to favor multiplication instead.

Metadata

- Monoid
- Sum monoid
- Concept

Concept: Total order

Headline

A transitive, antisymmetric, and total (binary) relation on some set

Illustration

In programming, total order serves for comparison of values. For instance, inLanguage:Haskell, we may leverage total order on numbers as follows:

```
> 41 < 42
True
> max 41 42
42
```

Some form of polymorphism may be used in many programming languages to define such a comparison-relate total order on given data types. For instance, in Haskell, there is a type class *Ord* for total order; its key member is *compare* which returns either LT, EQ, or GT. For instance:

```
> compare 41 42
LT
```

Let us illustrate the definition a total order for a simple data type for natural numbers:

```
-- Peano natural numbers
data Nat = Zero | Succ Nat
```

Before we define a total order for natural numbers, let us defineequality, as it is effectively a precondition for total order. In Haskell, we instantiate the type class *Eq* hence:

```
-- Equality of natural numbers
instance Eq Nat
  where
    Zero == Zero = True
    Zero == (Succ _) = False
    (Succ _) == Zero = False
    (Succ x) == (Succ y) = x == y
```

Thus, all pairs of constructor patterns are examined and accordingly mapped to truth values while subterms are processed recursively, when necessary. We can test for equality as follows:

```
> Succ Zero == Zero
False
> Succ Zero == Succ Zero
True
```

The type-class instance for total order follows the same scheme:

```
-- Total order on natural numbers
instance Ord Nat
```

```
 where
  compare Zero Zero = EQ
  compare Zero (Succ _) = LT
  compare (Succ _) Zero = GT
  compare (Succ x) (Succ y) = compare x y
```

We can test for total order as follows:

```
> compare (Succ Zero) Zero
GT
> compare (Succ Zero) (Succ Zero)
EQ
```

Metadata

- http://en.wikipedia.org/wiki/Total_order
- http://mathworld.wolfram.com/TotalOrder.html
- Concept

Concept: Type-class polymorphism

Headline

A form of bounded polymorphism based on type classes as in Language:Haskell

Illustration

See type class.

Metadata

- http://en.wikipedia.org/wiki/Type_class
- Bounded polymorphism
- Ad-hoc polymorphism

Concept: Type class

Headline

An abstraction mechanism for bounded polymorphism

Illustration

Type classes are not to be confused with OO classes. In fact, type classes may be somewhat compared with OO interfaces. Type classes have been popularized by Haskell. Similar constructs exist in a few other languages. Type classes capture operations that may be defined for many types. The operations can be defined differently for each type, i.e., for each instance of a type class.

All subsequent illustrations leverage Haskell. Let us consider the following datatypes of bits and bitstreams which represent unsigned binary numbers. We are going to enrich these datatypes with some functionality eventually, with the help of type classes:

```
-- A bit can be zero or one
data Bit = Zero | One

-- Bit streams of any length
newtype Bits = Bits { getBits :: [Bit] }
```

Thus, the binary number "101" would be represented as follows:

```
Bits [One,Zero,One]
```

Now suppose that we want to define some standard operations for bits and bitstreams: equality, total order, unparsing to text, parsing from text, and possibly others. Let us begin with unparsing (conversion) to text. To this end, we should implement Haskell's type-class-polymorphic function *show* so that it produces text like this:

```
> show (Bits [One,Zero,One])
"101"
```

Here is the type class *Show* which declares indeed the polymorphic *show* function:

```
class Show a
  where
    show :: a -> String
```

In reality, the type class has not just one member, *show*, as shown, but we omit the discussion of the other members here for brevity. The type class is parameterized in a type *a* for the actual type for which to implement the members. Here are the type-class instances for bits and bit streams:

```
-- Show bits
instance Show Bit
  where
```

```
    show Zero = "0"
    show One = "1"

-- Show bit streams
instance Show Bits
  where
    show = concat . map show . getBits
```

Thus, the instance fills the position of the type parameter with an actual type such as *Bit* and *Bits*. Also, the member function *show* is actually defined, while assuming the specific type. We show a bit as either "0" or "1". We show a bit stream by showing all the individual bits and concatenating the results.

The inverse of *show* is *read*. There is also a corresponding type class *Read*, which we skip here for brevity. Let us consider equality instead. There is again a type class which captures the potential of equality for many types:

```
class Eq a
  where
    (==) :: a -> a -> Bool
```

The member "(==)" is the infix operation for testing two bit streams to be equal. Arguably, bit streams are equal, if they are of the same length and they agree on each other bit by bit. In fact, the following definition is a bit more general in that it also trims away preceding zero bits:

```
-- Test bits for equality
instance Eq Bit
  where
    Zero == Zero = True
    Zero == One = False
    One == One = True
    One == Zero = False

-- Test bit streams for equality
instance Eq Bits
  where
    x == y =  length x' == length y'
           && and (map (uncurry (==)) (zip x' y'))
      where
        x' = trim (getBits x)
        y' = trim (getBits y)
        trim [] = []
        trim z@(One: ) = z
        trim (Zero:z) = trim z
```

For instance:

```
-- Test bit streams for equality
> let b101 = read "101" :: Bits
> let b0101 = read "0101" :: Bits
> let b1101 = read "1101" :: Bits
> b101 == b0101
True
> b101 == b1101
False
```

Actually, bit streams are (unsigned) binary numbers. Thus, we should also instantiate the corresponding type classes for number types. Operations on number types are grouped in multiple type classes. The type class *Num* deals with addition, subtraction, multiplication, and a few other operations, but notably no division:

```
class (Eq a, Show a) => Num a
  where
    (+) :: a -> a -> a
    (*) :: a -> a -> a
    (-) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a
    signum :: a -> a
    fromInteger :: Integer -> a
```

We would like to instantiate the *Num* type class for bit streams. There are different ways of doing this. For instance, we could define addition by bitwise addition, right at the level of bit streams, or we could instead resort to existing number types. For simplicity, we do indeed conversions from and to *Integer*, in fact, any *integral* type:

```
-- Convert bits to an integer
bits2integral :: Integral a => Bits -> a
bits2integral = foldl f 0 . getBits
  where
    f a b = a * 2 + (bit2int b)
    bit2int Zero = 0
    bit2int One = 1

-- Convert a (non-negative) integral to bits
integral2bits :: Integral a => a -> Bits
integral2bits i | i < 0 = error "Bits are unsigned"
integral2bits i = Bits (f [] i)
  where
    f xs 0 = xs
    f xs i = f (x:xs) (i `div` 2)
      where
        x = if odd i then One else Zero
```

On these grounds, we can trivially instantiate the *Num* type class for *Bits* by simply reusing the existing instance for Integer through systematic conversions.

```
-- Bits as a Num type
instance Num Bits
  where
    x + y = integral2bits z'
      where
        x' = bits2integral x
        y' = bits2integral y
        z' = x' + y'
    x * y = integral2bits z'
      where
        x' = bits2integral x
        y' = bits2integral y
        z' = x' * y'
    x - y = integral2bits z'
      where
        x' = bits2integral x
        y' = bits2integral y
        z' = x' - y'
```

```
  abs = id
  signum = integral2bits
       . signum
       . bits2integral
  fromInteger = integral2bits
```

The examples given so far are concerned with predefined type classes. However, type classes can also be declared by programmers in their projects. Let's assume that we may need to convert data from different formats into ``Int*s. Here is a corresponding type class with a few instances:*

```
class ToInt a
 where
  toInt :: a -> Maybe Int

instance ToInt Int
 where
  toInt = Just

instance ToInt Float
 where
  toInt = Just . round

instance ToInt String
 where
  toInt s =
    case reads s of
     [(i, "")] -> Just i
      _ -> Nothing
```

The conversion can be illustrated like this:

```
*Main> toInt "5"
Just 5
*Main> toInt "foo"
Nothing
*Main> toInt (5::Int)
Just 5
*Main> toInt (5.5::Float)
Just 6
```

In Haskell, type-class parameters are not limited to types, but, in fact, type classes may be parameterized in type constructors. Consider the following type class which models different notions of size for container types:

```
-- Notions of size for container types
class Size f
 where
  -- Number of constructors
  consSize :: f a -> Int
  -- Number of elements
  elemSize :: f a -> Int
```

Here is a straightforward instance for lists:

```
instance Size []
 where
  consSize = (+1) . length
```

```
  elemSize = length
```

Let's also consider sizes forrose trees:

```
-- Node-labeled rose trees
data NLTree a = NLTree a [NLTree a]
 deriving (Eq, Show, Read)

instance Size NLTree
 where
  consSize (NLTree _ ts) =
     1
   + consSize ts
   + sum (map consSize ts)
  elemSize (NLTree _ ts) =
     1
   + sum (map elemSize ts)

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

instance Size LLTree
 where
  consSize (Leaf _) = 1
  consSize (Fork ts) =
     consSize ts
   + sum (map consSize ts)
  elemSize (Leaf _) = 1
  elemSize (Fork ts) =
   sum (map elemSize ts)
```

A few illustrations are due:

```
*Main> let list = [1,2,3]
*Main> let nltree = NLTree 1 [NLTree 2 [], NLTree 3 []]
*Main> let lltree = Fork [Leaf 1, Fork [Leaf 2, Leaf 3]]
*Main> consSize list
4
*Main> elemSize list
3
*Main> consSize nltree
8
*Main> elemSize nltree
3
*Main> consSize lltree
9
*Main> elemSize lltree
3
```

Metadata

- http://en.wikipedia.org/wiki/Type class
- Abstraction mechanism
- Vocabulary:Haskell
- http://www.haskell.org/tutorial/classes.html
- Document:LaemmelO06

- Resource:Haskell%27s overlooked object system

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 treesfunctors 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])
```

Metadata

- http://en.wikipedia.org/wiki/Rose_Tree
- http://www.haskell.org/haskellwiki/Algebraic_data_type#Rose_tree
- Data structure
- Vocabulary:Functional programming

Concept: Type-class instance

Headline

Type-specific function definitions for a type class

Illustration

See the concept of type classes for an illustration.

Metadata

- Vocabulary:Haskell
- Concept
