Script: Data modeling in Haskell

Headline

Basic data modeling techniques in Haskell

Description

Basic concepts of data modeling in Haskell are covered. One important aspect of data modeling is the choice between structural versus nominal typing. This distinction gives rise to Haskell's type synonyms, algebraic data types, and record types. These options are conveniently illustrated with different data models for the system:Company. Another important aspect is the choice between different modeling options for recursive data structures, specifically the use of data composition and data variation. These options map to certain idioms of using algebraic data types in Haskell. These options are conveniently illustrated with different data models for the system:Company, when departmental nesting is taken into account.

Concepts

- Structural typing
- Nominal typing
- Type synonym
- Newtype
- Algebraic data type
- Data constructor
- Constructor component
- Pattern matching
- Case expression
- Record type
- Data composition
- Data variation
- Type constructor
- Tuple type
- Either type
- List type
- Maybe type

Languages

- Language:Haskell

Features

- Feature:Total
- Feature:Cut

Contributions

- Contribution:haskellEngineer
- Contribution:haskellData
- Contribution:haskellRecord
- Contribution:haskellComposition
- Contribution:haskellVariation

Metadata

- Course:Lambdas in Koblenz
- Script:Searching and sorting in Haskell

System: Company

Headline

An imaginary HRMS system

Description

System:Company is an imaginary 'Human resource management system (HRMS)' (i.e., an information system) implementations of which ('contributions') are documented on 101wiki. The system is supposed to model the company structure in terms of employees and possibly the hierarchical structure of departments. Employees are modeled in terms of their names, addresses, salaries, and possibly additional properties. The system is supposed to meet certain functional requirements such as totaling all salaries in the company. The system may also be subjected to non-functional requirements such as persistence or distribution. Features are not collected for the sake of an interesting HRMS system. Instead, features are designed to exercise interesting characteristics of software languages and software technologies. Most features are optional so that contributions have the freedom of choice to focus on features that are particularly interesting for a certain objective of language or technology demonstration.

There are the following features:

- Company: Companies, department, employees
- Total: Total the salaries of employees
- Median: Compute the median of the salaries
- Cut: Cut the salaries of employees in half
- Depth: Compute nesting depth of departments
- COI: Conflicts of interests for employees
- Mentoring: Associate mentors and mentees
- Ranking: Enforce salary to correlate with ranks
- Singleton: Constrain for a single company
- History: Maintain and analyze company history
- Serialization: De-/serialize companies
- Persistence: Persist companies
- Mapping: Map companies across technological space
- Distribution: Distribute companies
- Parallelism: Total or cut in parallel
- Logging: Log company changes
- Browsing: Browse companies interactively
- Editing: Edit companies interactively
- Restructuring: Restructure companies interactively
- Web UI: Operate on companies in a web browser
- Parsing: Parse companies in concrete syntax
- Unparsing: Pretty print companies

The set of all features can also be arranged in a feature model as defined by the following constraints:

- Data requirements
  - Feature:Company (XOR)
    - Feature:Hierarchical company
    - Feature:Flat company
  - Feature:COI?
  - Feature:Mentoring?
  - Feature:Ranking?
  - Feature:Singleton?
  - Feature:History?
- Functional requirements
  - Feature:Total?
  - Feature:Cut?
  - Feature:Median?
  - Feature:Logging?
  - Feature:Depth?
  - Feature:Parsing?
  - Feature:Unparsing?
  - Feature:History?
- Non-functional requirements
  - Feature:Serialization? (XOR)
    - Feature:Open serialization
    - Feature:Closed serialization
  - Feature:Persistence?
  - Feature:Mapping?
  - Feature:Distribution?
  - Feature:Parallelism? (OR)
    - Feature:Data parallelism
    - Feature:Task parallelism
- UI requirements
  - Feature:Browsing?
  - Feature:Editing?
  - Feature:Undo-redo?

- Feature:Restructuring?
- Feature:Web UI?

- More constraints
  - Feature:Depth => Feature:Hierarchical company
  - Feature:Ranking => Feature:Hierarchical company
  - Feature:History => Feature:Total
  - Feature:History => Feature:Median
  - Feature:Web UI => Feature:Browsing
  - Feature:Editing => Feature:Browsing
  - Feature:Restructuring => Feature:Editing
- Emerging and vanishing features
  - ~~Feature:Reporting~~
  - ~~Feature:Charting~~
  - ~~Feature:Grouping~~
  - ~~Feature:Gender~~
  - ~~Feature:Bonus~~
  - ~~Feature:Job description => Feature:Job role~~
  - ~~Feature:Annual employee review~~

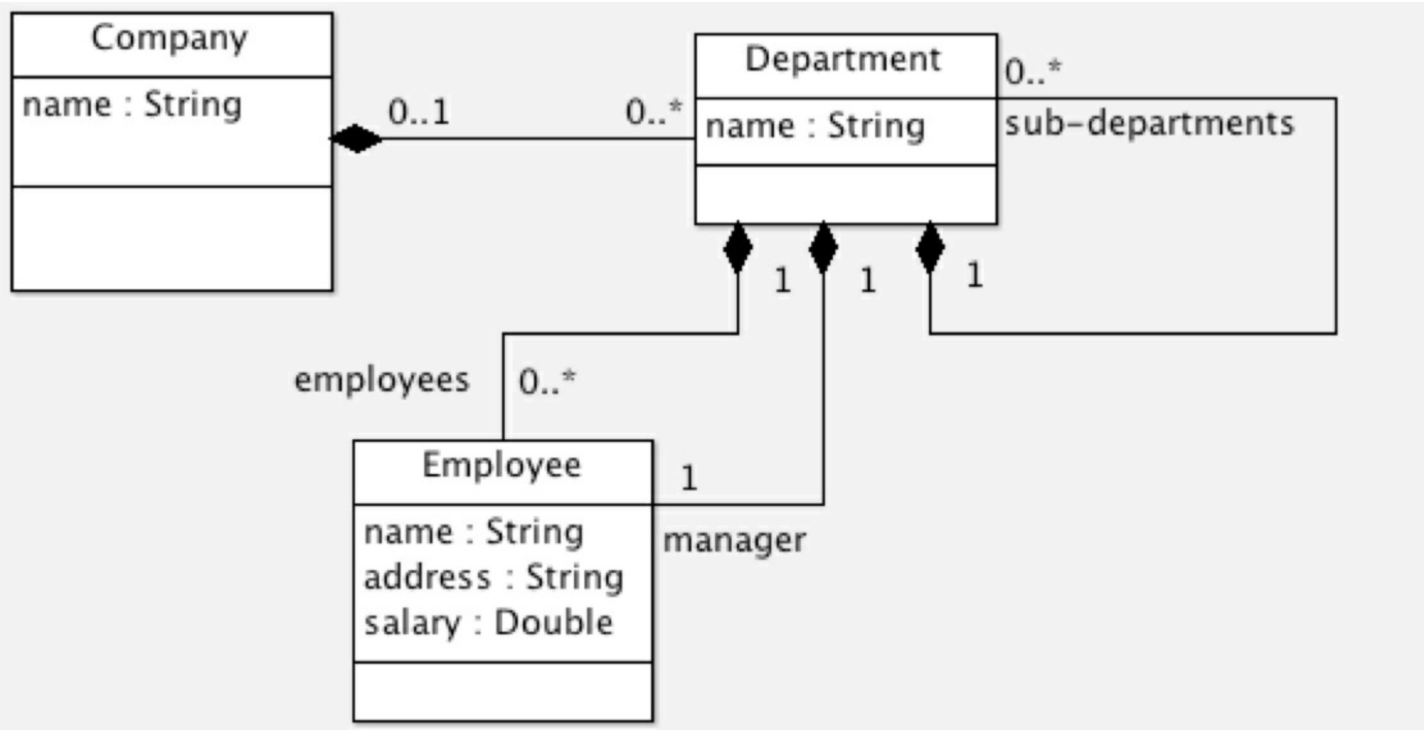
This specification is under construction.

We use the following informal notation here:

- *f*? means that the feature *f* is optional.
- *f* (OR) means that *f* is an OR feature; any operands may be chosen, but at least one, unless *f* is optional.
- *f* (XOR) means that *f* is an XOR feature; either of its operands must be selected, but not several of them.
- *f1* => *f2* means that if *f1* is selected then *f2* must be selected.
- ~~*f*~~ (i.e., *f* with strikethrough) means that the feature is only emerging or already vanishing.

Illustration

The following UML class diagram models the basic structure of the system.



See Theme:Starter for a few very simple contributions in varying languages. These are mostly implementations of the system in varying programming languages, but a UML-based model (as shown above) is also included.

Metadata

- Human resource management system
- Namespace:Feature

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: Constructor component

Headline

A component of a data constructor

Illustration

See the illustration for data constructors.

Metadata

- Vocabulary:Functional programming
- Concept

Concept: Type synonym

Headline

Abstraction over type expressions

Illustration

The name type synonym is specifically used in Language:Haskell. (The same concept goes by the name "typedef" in, for example, Language:C.) For instance, the following Haskell declaration introduces a type synonym for salaries to be represented as floats.

```
type Salary = Float
```

The choice of a type synonym implies that salaries and floats are compatible in a typing sense: any float is immediately acceptable whereever a salary is expected, and vice versa. The type synonym is merely a convenience without any proper effect on typing.

Thus, if you look at the signature of a function, such as total:

```
total :: Company -> Float
```

This signature could as well be transformed by resolving all type synonyms (described by Feature:Flat_company) to a less understandable variant:

```
total :: ([Char], [([Char], [Char], Float)]) -> Float
```

Metadata

- http://en.wikipedia.org/wiki/Typedef
- http://www.haskell.org/haskellwiki/Type_synonym
- Vocabulary:Functional programming
- Structural typing
- Concept

Contribution: haskellVariation

Headline

Data variation in Language:Haskell with algebraic data types

Characteristics

The data model leverages data variation for companies with departmental nesting. Thus, analgebraic data type is used for subunits of departments (i.e., employees and departments) so that recursive nesting can be expressed. The algebraic data type needs indeed two data constructors. Thus, data variation is exercised, but see Contribution:haskellComposition for an alternative without data variation.

Illustration

The data model leverages an algebraic data type for subunits of departments; in this manner recursion is enabled:

```
{-| 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, and sub-units
type Department = (Name, Manager, [SubUnit])

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

-- | A sub-unit is either an employee or a sub-department
data SubUnit = EUnit Employee | DUnit Department
 deriving (Eq, Read, Show)

-- | 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 sample company looks like this:

```
{- | 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",
    [
      ( "Research",
        ("Craig", "Redmond", 123456),
        [
          EUnit ("Erik", "Utrecht", 12345),
          EUnit ("Ralf", "Koblenz", 1234)
        ]
      ),
      ( "Development",
        ("Ray", "Redmond", 234567),
        [
          DUnit (
            "Dev1",
            ("Klaus", "Boston", 23456),
            [
              DUnit (
                "Dev1.1",
                ("Karl", "Riga", 2345),
                [EUnit ("Joe", "Wifi City", 2344)]
              )
            ]
          )
        ]
      )
    ]
  )
```

Feature:Total is implemented as follows:

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

```
module Company.Total where

import Company.Data

-- | Total all salaries in a company
total :: Company -> Float
total (_, ds) = sum (map totalDepartment ds)
 where
  -- Total salaries in a department
  totalDepartment :: Department -> Float
  totalDepartment (_, m, sus)
    = getSalary m
    + sum (map totalSubunit sus)
   where
    -- Total salaries in a subunit
    totalSubunit :: SubUnit -> Float
    totalSubunit (EUnit e) = getSalary e
    totalSubunit (DUnit d) = totalDepartment d

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

The following salary total is computed for the sample company:

```
399747.0
```

Relationships

See Contribution:haskellComposition for a contribution with a similar data model such that data variation is not exercised, but only data composition.

Architecture

See Contribution:haskellComposition.

Usage

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

Metadata

- Language:Haskell
- Language:Haskell 98
- Technology:GHC
- Technology:Cabal
- Feature:Hierarchical company
- Feature:Total
- Feature:Cut
- Feature:Closed serialization
- Contributor:rlaemmel
- Theme:Haskell data
- Theme:Haskell introduction
- Contribution:haskellComposition
- Contribution:haskellEngineer

Contribution: haskellComposition

Headline

Data composition in Haskell with algebraic data types

Characteristics

The data model leverages data composition for companies with departmental nesting. Thus, analgebraic data type is used for departments so that recursive nesting can be expressed. The algebraic data type only needs a single data constructor. Thus, data variation is not exercised, but see Contribution:haskellVariation for an alternative with data variation.

Illustration

The data model leverages an algebraic data type for departments; in this manner recursion is enabled:

```
{-| 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 sample company looks like this:

```
{- | 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)]
            ]
            []
        ]
        []
    ]
  )
```

Feature:Total is implemented as follows:

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

module Company.Total where

import Company.Data
```

```
-- | Total all salaries in a company
total :: Company -> Float
total (_, ds) = totalDepartments ds
 where

  -- Total salaries in a list of departments
  totalDepartments :: [Department] -> Float
  totalDepartments [] = 0
  totalDepartments (Department _ m ds es : ds')
    = getSalary m
    + totalDepartments ds
    + totalEmployees es
    + totalDepartments ds'
   where

    -- Total salaries in a list of employees
    totalEmployees :: [Employee] -> Float
    totalEmployees [] = 0
    totalEmployees (e:es)
     = getSalary e
     + totalEmployees es

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

The following salary total is computed for the sample company:

```
399747.0
```

Relationships

- See Contribution:haskellVariation for a contribution with a similar data model such thatdata variation is exercised in addition todata composition.
- See Contribution:haskellEngineer for a contribution with a simple data model without support for departmental nesting. No algebraic data types are leveraged.
- See Contribution:haskellData for a contribution with a simple data model without support for departmental nesting. Algebraic data types are leveraged systematically for all types to distinguish the types nominally.

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

-- | Total all salaries in a company
total :: Company -> Float
total (_, ds) = totalDepartments ds
  where

    -- Total salaries in a list of departments
    totalDepartments :: [Department] -> Float
    totalDepartments [] = 0
    totalDepartments (Department _ m ds es : ds')
        = getSalary m
        + totalDepartments ds
        + totalEmployees es
        + totalDepartments ds'
      where

        -- Total salaries in a list of employees
        totalEmployees :: [Employee] -> Float
        totalEmployees [] = 0
        totalEmployees (e:es)
          = getSalary e
          + totalEmployees es

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

: the implementation of Feature:Total

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

module Company.Cut where

import Company.Data

-- | Cut all salaries in a company
cut :: Company -> Company
cut (n, ds) = (n, (map cutD ds))
  where
    -- Cut all salaries in a department
    cutD :: Department -> Department
    cutD (Department n m ds es)
      = Department n (cutE m) (map cutD ds) (map cutE es)
      where
        -- Cut the salary of an employee in half
        cutE :: Employee -> Employee
        cutE (n, a, s) = (n, a, s/2)
```

: the implementation of Feature:Cut

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

module Main where

import Company.Data
import Company.Sample
import Company.Total
import Company.Cut
import Test.HUnit
import System.Exit

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

-- | Compare total after cut of sample company with baseline
cutTest = total sampleCompany / 2 ~=? total (cut sampleCompany)

-- | 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 "cut" cutTest,
    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
- Feature:Hierarchical company
- Feature:Total
- Feature:Cut
- Feature:Closed serialization
- Contributor:rlaemmel
- Theme:Haskell data
- Theme:Haskell introduction
- Contribution:haskellVariation
- Contribution:haskellEngineer

Contribution: haskellRecord

Headline

Use of record types in Language:Haskell

Characteristics

A data model for flat companies is defined in terms of Haskell's record types. Such record types are essentially algebraic data types. We only use record types for compound data. Otherwise, we use Haskell's newtypes, which is a special form of algebraic data type. Other than that, the contribution is a simple variation on Contribution:haskellData which uses plain algebraic data types for all types.

Illustration

The data model looks like this:

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

module Company.Data where

-- | A company consists of name and employee list
data Company = Company {
    getCompanyName :: Name,
    getEmployees :: [Employee]
  }
    deriving (Eq, Show, Read)

-- | An employee consists of name, address, and salary
data Employee = Employee {
    getEmployeeName :: Name,
    getAddress :: Address,
    getSalary :: Salary
  }
    deriving (Eq, Show, Read)

-- | Names as strings
newtype Name = Name String
  deriving (Eq, Show, Read)

-- | Addresses as strings
newtype Address = Address String
  deriving (Eq, Show, Read)

-- | Salaries as floats
newtype Salary = Salary { getFloat :: Float }
  deriving (Eq, Show, Read)
```

A sample company looks like this:

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

module Company.Sample where

import Company.Data

-- | A sample company useful for basic tests
sampleCompany :: Company
sampleCompany = Company {
  getCompanyName = Name "Acme Corporation",
  getEmployees = [
    Employee (Name "Craig") (Address "Redmond") (Salary 123456),
    Employee (Name "Erik") (Address "Utrecht") (Salary 12345),
    Employee (Name "Ralf") (Address "Koblenz") (Salary 1234),
    Employee (Name "Ray") (Address "Redmond") (Salary 234567),
    Employee (Name "Klaus") (Address "Boston") (Salary 23456),
    Employee (Name "Karl") (Address "Riga") (Salary 2345),
    Employee (Name "Joe") (Address "Wifi City") (Salary 2344)
  ]
}
```

Feature:Total is implemented as follows:

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

module Company.Total where

import Company.Data

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

-- | Extract all salaries in a company
salaries :: Company -> [Float]
salaries = getSalaries . getEmployees
  where
```

```
-- Extract all salaries of lists of employees
getSalaries :: [Employee] -> [Float]
getSalaries [] = []
getSalaries (e:es) = getFloat (getSalary e) : getSalaries es
```

Relationships

- The present contribution is a slightly more complex variation on Contribution:haskellEngineer in that it uses data types (in fact, record types) as opposed to type synonyms.
- See also Contribution:haskellData, which uses plain data types instead of record types.

Architecture

See Contribution:haskellEngineer.

Usage

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

Metadata

- Theme:Haskell data
- Language:Haskell
- Language:Haskell 98
- Technology:GHCi
- Feature:Flat company
- Feature:Total
- Feature:Cut
- Feature:Closed serialization
- Contributor:rlaemmel
- Contribution:haskellEngineer
- Contribution:haskellData
- Contribution:haskellEngineer

Concept: Type constructor

Headline

An abstraction for constructing new types

Illustration

For instance, in functional programming with Haskell, these are typical type constructor:

- The list type constructor for constructing list types from an element type
- Any tuple type constructor for constructing types of products from two component types
- The maybe type constructor for adding partiality to a type
- The either type constructor for combining types as cases

These types could or are defined in Haskell in follows:

```
data [a] = [] | (a:[a])
data (a, b) = (a, b)
data Maybe a = Nothing | Just a
data Either a b = Left a | Just a
```

If we were to remove the special notation for lists and tuples, thus using ordinary type and constructor names, then the first two declarations take this form:

```
data List a = Nil | Cons a (List a)
data Pair a b = Pair a b
```

Metadata

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

Concept: Algebraic data type

Headline

A type for the construction of terms

Illustration

Algebraic data types are typically supported by functional programming languages. For instance, Language:Haskell, Language:Scala, and Language:SML support algebraic data types. Illustrations are given here for Language:Haskell.

A data type for shapes can be defined as follows:

```
data Shape = Circle Float
        | Rectangle Float Float
```

The data contructor *Circle* serves for the representation of circles; the one and only constructor component serves for the radius. The data constructor *Rectangle* serves for the representation of rectangles; the two constructor components serve for width and height.

Constructors can be used as functions to construct terms:

```
myCircle :: Shape
myCircle = Circle 42

myRectangle :: Shape
myRectangle = Rectangle 77 88
```

In fact, constructors are functions with the types of the constructor components as argument types and the type of algebraic data type as the result type:

```
> :t Circle
Circle :: Float -> Shape
> :t Rectangle
Rectangle :: Float -> Float -> Shape
```

Pattern matching can be applied to terms of algebraic data types:

```
-- Test whether the shape is a circle
isCircle (Circle _) = True
isCircle (Rectangle _ _) = False
```

See Contribution:haskellData for a more profound illustration of algebraic data types.

All predefined, compound types of Haskell are essentially algebraic data types. For instance, Haskell's Booleans could be defined by an algebraic data type with two data constructors as follows:

```
data Bool = True | False
```

Haskell's lists could be defined by an algebraic data type with two construtors as follows:

```
data List a = Nil | Cons a (List a)
```

The constructors *Nil* and *Cons* are meant to correspond to the empty list and ":" of Haskell's built-in lists. The *Nil* constructor has no constructor component, whereas the *Cons* constructor has two constructor components.

See also Maybe types for yet another illustration of algebraic data types.

Metadata

- http://en.wikipedia.org/wiki/Algebraic_data_type
- http://www.haskell.org/haskellwiki/Algebraic_data_type
- Type
- Vocabulary:Functional programming

Concept: Nominal typing

Headline

Equivalence of types based on their names

Illustration

See structural typing for illustration.

Metadata

- http://en.wikipedia.org/wiki/Nominative_type_system
- http://c2.com/cgi/wiki?NominativeAndStructuralTyping
- Structural typing

Concept: Pattern matching

Headline

Matching values against patterns to bind variables

Description

Pattern matching may be concerned with different kinds of types, e.g.,text or trees. In the case of text,regular expressions provide the foundation for patterns. In the case of trees and specifically in the context of functional programming, algebraic data types provide the foundation for patterns; in this case, pattern matching is concerned with case discrimination on different constructor patterns such that variables are bound in successfully matched patterns for use in expressions.

Illustration

Pattern matching in Haskell

The basics of Haskell's pattern matching are very similar to those of other functional programming languages.

Pattern matching on pairs

```
-- Project a pair to first component
fst :: (a,b) -> a
fst (x,_) = x

-- Project a pair to second component
snd :: (a,b) -> b
snd (_,x) = x
```

These two functions *fst* and *snd* are defined like this (or similarly) in thePrelude module of Language:Haskell. They are defined by pattern matching on the structure of tuples; see the the left-hand sides of the function definitions. The idea of such pattern matching is of course that variables in the pattern (on the left-hand side) can be used in the expression of the definition (on the right-hand side).

Pattern matching on lists

```
-- Retrieve head (first element) of a list
head :: [a] -> a
head (x:_) = x

-- Retrieve tail (all but first element) of a list
tail :: [a] -> [a]
tail (_:xs) = xs
```

These two functions *head* and *tail* are defined like this (or similarly) in thePrelude module of Language:Haskell. They demonstrate that non-empty lists are constructed with the cons constructor ":" from a head and a tail.

Pattern matching is particularly convenient, when functions should be defined by case discrimination on the different constructor patterns for a data type. Consider, for example, the length function (again borrowed from the Prelude); this definition consists of two equations: one for the case of an empty list and another case for non-empty lists:

```
-- Determine length of list
length :: [a] -> Int
length [] = 0
length (_:xs) = length xs + 1
```

Other forms of pattern matching

- Pattern matching is particularly useful for user-defined algebraic data types.
- Pattern matching is not limited to the use on left-hand sides of equations. Instead, pattern matching can also be performed throughcase expressions in an expression context.
- Haskell patterns may involve so-called guards to control the selection of equations (cases) not just purely on the grounds structure but also computations on top of bound variables.
- Haskell provides different forms of patterns to deal withlaziness. This is not further discussed here.

Metadata

- Vocabulary:Functional programming
- Vocabulary:Data
- http://en.wikipedia.org/wiki/Pattern_matching
- http://en.wikibooks.org/wiki/Haskell/Pattern_matching

Contribution: haskellEngineer

Headline

Basic software engineering for Haskell

Characteristics

The contribution demonstrates basic means of modularization (using Haskell's native module system), code organization (using where clauses for local scope), packaging (using Technology:Cabal), documentation (using Technology:Haddock), and unit testing (using Technology:HUnit). Other than that, only basic language constructs are exercised and a very limited feature set of the system:Company is implemented. The contribution is indeed more of a showcase for a pattern for modularization, code organization, packaging, documentation, and unit testing.

Illustration

Modular organization

The contribution consists of the modules as listed in following file:

```
name:           haskellEngineer
version:        0.1.0.0
synopsis:       Basic software engineering for Haskell
homepage:       http://101companies.org/wiki/Contribution:haskellEngineer
build-type:     Simple
cabal-version:  >=1.9.2

library
  exposed-modules:
    Main
    Company.Data
    Company.Sample
    Company.Total
    Company.Cut
  build-depends:    base >=4.4 && < 5.0, HUnit
  hs-source-dirs:   src

test-suite basic-tests
    main-is:        Main.hs
    build-depends:  base, HUnit
    hs-source-dirs: src
    type:           exitcode-stdio-1.0
```

The modules implement features as follows:

- Company/Data.hs: Feature:Flat company.
- Company/Sample.hs: A sample company.
- Company/Total.hs: Feature:Total.
- Company/Cut.hs: Feature:Cut.
- Main.hs: Unit tests for demonstration.

For instance, the implementation of Feature:Total takes this form:

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

module Company.Total where

import Company.Data

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

    -- Extract all salaries in a company
    salaries :: Company -> [Salary]
    salaries (_, es) = getSalaries es
      where

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

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

Please note how "where clauses" are used to organize the declarations in such a way that it is expressed what function is a helper function to what other function. The declaration of such local scope also implies that the helper functions do not feed into the interface of the module.

Haddock comments

Technology:Haddock comments are used to enable documentation generation. Consider again the module shown above. Haddock comments are used for the functions *total* and *salaries* but not for the remaining functions, as they are not exported and thus, they do not need to be covered by the generated documentation.

External dependencies

The contribution has the following dependencies; see again the .cabal file:

```
build-depends: base >=4.4 && < 5.0, HUnit
```

These packages serve the following purposes:

- base: This is the Haskell base package; a range of versions is permitted.
- HUnit: This is the package for Technology:HUnit; its version is not explicitly constrained.

HUnit testcases

The contribution is tested by the following test cases:

```
tests =
 TestList [
  TestLabel "total" totalTest,
  TestLabel "cut" cutTest,
  TestLabel "serialization" serializationTest
 ]
```

For instance, the test case for serialization looks as follows:

```
serializationTest = sampleCompany ~=? read (show sampleCompany)
```

Relationships

- The present contribution is an "engineered" variation on Contribution:haskellStarter. That is, modularization, packaging, documentation, and unit testing was applied.
- Several other contributions derive from the present contribution more or less directly by demonstrating additional language or technology capabilities or implementing additional features of the system:Company.

Architecture

Modules to feature mapping:

- Company.Data: Feature:Flat company
- Company.Sample: A sample company
- Company.Total: Feature:Total
- Company.Cut: Feature:Cut
- Main: Unit tests for demonstration

Usage

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

Metadata

- Language:Haskell
- Language:Haskell 98
- Technology:GHC
- Technology:Cabal
- Technology:HUnit
- Technology:Haddock
- Feature:Flat company
- Feature:Closed serialization
- Feature:Total
- Feature:Cut
- Contributor:rlaemmel
- Theme:Haskell introduction
- Contribution:HaskellStarter

Concept: Data constructor

Headline

A constructor of an algebraic data type

Illustration

See the illustration for algebraic data types.

Metadata

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

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.

Metadata

- http://www.thefreedictionary.com/salary+cut
- Functional requirement
- Transformation
- Type-preserving transformation
- Iterator-based transformation
- Optional feature
- Type-preserving transformation

Concept: Data variation

Headline

Construction of data according to variants

Note

Data variation is not an established term, but it naturally arises as a generalization and dualization of existing notions that are used in data modeling and programming. Common forms of data variation are based on variant types and type generalization. For clarity, the general term data variation is used on this wiki, whenever appropriate.

Description

Data variation entails data variants, of which one must be chosen when actual data is constructed. The data variants may be specified, for example, as a variant type or as OO types related to a common base type through type generalization. Data variation may be seen as a principle means of going beyond data composition.

Illustration

The following Language:Haskell-based data model for the @system leverages data variation in one spot and otherwise data composition. The data model is based on algebraic data types. The data type for subunits declare two constructors to model different types of subunits, as needed for aggregating subunits of departments.

```
data Company = Company Name [Department]
data Department = Department Name Manager [SubUnit]
data Employee = Employee Name Address Salary
data SubUnit = EUnit Employee | DUnit Department
type Manager = Employee
type Name = String
type Address = String
type Salary = Float
```

The snippet originates from Contribution:haskellVariation.

Metadata

- Vocabulary:Data modeling
- Vocabulary:Programming
- Data composition
- Concept

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: Newtype

Headline

A special form of algebraic data type in Language:Haskell

Illustration

Consider the following declaration of a salary type:

```
type Salary = Float
```

This declaration introduces merely a type synonym, but it enforces no type distinction. Floats and salaries are compatible in the sense of structural typing. If we were to enforce a type distinction, then the following type declaration could be preferred instead:

```
data Salary = Salary Float
```

Thus, salaries and floats are no longer compatible at a typing level; a float may be "wrapped" as a salary; a salary may be "unwrapped" to retrieve a float. Indeed, this special case of using algebraic data types just for making type distinctions is specifically supported by newtypes in Haskell. Accordingly, the following type declaration uses a newtype:

```
newtype Salary = Salary Float
```

Syntactically, a newtype is an algebraic data type with only one data constructor with in turn only one constructor component. Semantically, this restriction implies that we can think of the constructor as serving for type distinction only without any semantical purpose such as grouping data.

Consider this program:

```
data X = X ()
newtype Y = Y ()
f (X _) = True
g (Y _) = True
```

When *f* is applied to *undefined*, then an exception is thrown, as proper pattern matching (term deconstruction) has to be performed in order to confirm the equation. When *g* is applied to *undefined*, then the equation is soundly applied (such that *True*) is returned because no pattern has to be matched and the undefined argument of *Y* is not inspected.

```
*Main> f undefined
*** Exception: Prelude.undefined
*Main> g undefined
True
*Main> f (X undefined)
True
```

Metadata

- Algebraic data type
- http://www.haskell.org/haskellwiki/Newtype
- Concept

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

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: Data composition

Headline

Composition of compound data from parts

Note

Data composition is not an established term, but it naturally arises as a generalization of existing notions that are used in data modeling and programming. A common form of data composition is object composition, which is actually sometimes also defined in a broad enough sense, not to be limited to objects in the sense of OO programming. For clarity, the general term data composition is used on this wiki, whenever appropriate.

Description

Data composition entails component entities (e.g., primitive data or previously composed data) and compound entities (e.g. objects, tuples, or records). Composition means that the components (say, the parts) are combined to form a compound (say, a whole). A principle means of going beyond data composition is to leverage data variation.

Illustration

The following Language:Haskell-based data model for the @system leverages data composition systematically. The data model is based on algebraic data types. The data types for companies, departments, and employees declare a single constructor to serve for data composition. Basic types for numbers and strings are used for some components. List types are also used.

```
data Company = Company Name [Department]
data Department = Department Name Manager [Department] [Employee]
data Employee = Employee Name Address Salary
type Manager = Employee
type Name = String
type Address = String
type Salary = Float
```

The snippet originates from Contribution:haskellComposition.

Metadata

- Composition
- Vocabulary:Data modeling
- Vocabulary:Programming
- Object composition
- Data variation

Concept: Tuple type

Headline

A data type for tuples

Illustration

We illustrate here the Haskell approach to tuple types.

We can form tuples of different length: pairs, triples, quadruples, ...:

```
Prelude> (1,"2")
(1,"2")
Prelude> (1,"2",True)
(1,"2",True)
Prelude> (1,"2",True,42.0)
(1,"2",True,42.0)
```

These tuples are of different tuple types:

```
Prelude> (1,"2") :: (Int,String)
(1,"2")
Prelude> (1,"2",True) :: (Int,String,Bool)
(1,"2",True)
Prelude> (1,"2",True,42.0) :: (Int,String,Bool,Float)
(1,"2",True,42.0)
```

We can think of these tuple types as being defined aspolymorphic types like this:

```
type (a,b) = (a,b)
type (a,b,c) = (a,b,c)
type (a,b,c,d) = (a,b,c,d)
```

Specific tuple types can also be expressed by other means than a designated type constructor for such types. For instance, the illustrative tuple types given above could also be declared like this:

```
data TupleType1 = TupleType1 Int String
data TupleType2 = TupleType2 Int String Bool
data TupleType3 = TupleType3 Int String Bool Float
```

(We reuse type names as constructor symbols here, which is possible in Haskell, as these are separate namespaces.) The advantage of the type type constructors (of different arities) is that they capture universally (polymorphically) the notion of ordered tuples, that is, for arbitrary operand types.

Metadata

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

Concept: Structural typing

Headline

Equivalence of types based on their structure

Illustration

Consider the following declarations in Language:Haskell:

```
type Point = (Float, Float) -- A Cartesian point
type Rectangle = (Float, Float) -- A rectangle with width and height
```

Because both types are defined as type synonyms, they are subject to structural typing. Thus, points and rectangles are compatible such that a point can be used whereever a rectangle is expected, and vice versa. This is arguably not intended. We may instead define the types as follows:

```
data Point = Point Float Float -- A Cartesian point
data Rectangle = Rectangle Float Float -- A rectangle with width and height
```

While both types are structurally equivalent, as both types declare one data constructor with the exact same types for the constructor components, the types are still different as nominal typing applies for Haskell's algebraic data types.

Metadata

- http://en.wikipedia.org/wiki/Structural_type_system
- http://c2.com/cgi/wiki?NominativeAndStructuralTyping
- Nominal typing
- Concept

Concept: Record type

Headline

A type of records

Illustration

Record types are available or conveniently expressible in many programming languages.

Record types in Haskell

Language:Haskell provides syntactic sugar for algebraic data types such that constructor components can be be labeled so that they can be accessed in a record-like manner. Consider, for example, the following algebraic data type for points:

```
data Point = Point Float Float
```

The data constructor can be defined using record notation instead:

```
data Point = Point { getX :: Float, getY :: Float }
```

Here is an example of constructing a record:

```
myPoint :: Point
myPoint = Point { getX = 42, getY = 88 }
```

Here is an example of accessing record components:

```
-- Compute the distance between two points
distance :: Point -> Point -> Float
distance p1 p2 = sqrt (deltax^2+deltay^2)
  where
    deltax = abs (getX p1 - getX p2)
    deltay = abs (getY p1 - getY p2)
```

The constructors and component selectors are of these types:

```
> :t Point
Point :: Float -> Float -> Point
> :t getX
getX :: Point -> Float
> :t getY
getY :: Point -> Float
```

We can also update records in the sense that we can construct new records from existing records by updating specific components. For instance:

```
myPoint' :: Point
myPoint' = myPoint { getY = 77 }
```

For what it matters, the position-based approach to construction, as with normal algebraic data types, can also be used. Thus, component selectors can be omitted at will. This is demonstrated with constructing the same point as above:

```
myPoint :: Point
myPoint = Point 42 88
```

Component selectors are also omitted during pattern matching:

```
-- Represent point as pair
toPair :: Point -> (Float, Float)
toPair (Point x y) = (x, y)
```

The record notation can also be used in algebraic data types with multiple constructors, e.g.:

```
data Shape = Circle { getRadius :: Float }
           | Rectangle { getWidth :: Float, getHeight :: Float }
```

Metadata

- http://en.wikipedia.org/wiki/Record_(computer_science)
- Type
- Vocabulary:Data structure

Contribution: haskellData

Headline

Use of algebraic data types in Language:Haskell

Characteristics

A data model for flat companies is defined in terms of Haskell's algebraic data types. Other than that, the contribution is a simple variation on Contribution:haskellEngineer. The systematic use of algebraic data types implies nominal type distinctions in the sense of nominal typing. For instance, arbitrary floats cannot be confused with salaries which are floats only structurally. The different kinds of names for companies, departments, and employees are not distinguished, even though this would also be possible, in principle.

Illustration

The data model looks like this:

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

module Company.Data where

-- | A company consists of name and employee list
data Company = Company Name [Employee]
  deriving (Eq, Show, Read)

-- | An employee consists of name, address, and salary
data Employee = Employee Name Address Salary
  deriving (Eq, Show, Read)

-- | Names as strings
data Name = Name String
  deriving (Eq, Show, Read)

-- | Addresses as strings
data Address = Address String
  deriving (Eq, Show, Read)

-- | Salaries as floats
data Salary = Salary Float
  deriving (Eq, Show, Read)
```

A sample company looks like this:

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

module Company.Sample where

import Company.Data

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

Feature:Total is implemented as follows:

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

module Company.Total where

import Company.Data

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

-- | Extract all salaries in a company
salaries :: Company -> [Float]
salaries (Company _ es) = getSalaries es
  where

    -- Extract all salaries of lists of employees
    getSalaries :: [Employee] -> [Float]
    getSalaries [] = []
    getSalaries (e:es) = getSalary e : getSalaries es
      where

        -- Extract the salary from an employee
```

```
getSalary :: Employee -> Float
getSalary (Employee _ _ (Salary s)) = s
```

Relationships

- The present contribution is a slightly more complex variation on Contribution:haskellEngineer in that it uses data types as opposed to type synonyms.
- See also Contribution:haskellRecord, which uses record types instead of plain data types.

Architecture

See Contribution:haskellEngineer.

Usage

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

Metadata

- Theme:Haskell data
- Language:Haskell
- Language:Haskell 98
- Technology:GHCi
- Feature:Flat company
- Feature:Total
- Feature:Cut
- Feature:Closed serialization
- Contributor:rlaemmel
- Contribution:haskellEngineer
- Contribution:haskellRecord
- Contribution:haskellEngineer

Concept: Case expression

Headline

An expression form to discriminate between different results

Illustration

Case expressions are typically available in functional programming languages as a means to perform pattern matching over values of algebraic data types. For instance, consider the following function in Language:Haskell:

```
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs
```

This definition expresses case discrimination in terms of multiple equations, but we could define a variation on *length* so that case discrimination is expressed by a single expression instead:

```
length' :: [a] -> Int
length' l =
 case l of
  [] -> 0
  (_:xs) -> 1 + length' xs
```

A case expression may feature multiple branches in the same way as a function definition may feature multiple equations. In essence, the syntax of having multiple equations is "syntactic sugar*; we can always suffice with function definitions with just one equation and reside to case expressions for pattern matching. Also, case expression is more general in so far that it is an expression form which does not require the introduction of an explicit function name.*

Metadata

- http://zvon.org/other/haskell/Outputsyntax/caseQexpressions_reference.html

---

Concept: Either type

Headline

A type for disjoint (indexed) sums over types

Illustration

We illustrate here the Haskell approach to either types.

The corresponding polymorphic type type constructor is defined as follows:

```
data Either a b = Left a | Right b
```

Thus, a value of an either type is either of one type or another and the choice is also conveyed by the constructors *Left* versus *Right*. One typical application scenario is error handling where one argument type models error messages (e.g., String) and the other argument type models successful results. In this instance, either types generalize maybe types.

Another typical application scenario is mixed-type computations. For instance, assume that we have some mathematical operations that may return both *Int* and *Float*. Here is a corresponding either type:

```
type IntOrFloat = Either Int Float
```

As an example of a function that needs to manipulate values of the either type, consider the following function that extracts a *Float* by applying the conversion *fromIntegral* if given an *Int*:

```
asFloat :: IntOrFloat -> Float
asFloat (Left x) = fromIntegral x
asFloat (Right x) = x
```

For instance:

```
> asFloat (Left 42)
42.0
> asFloat (Right 42.0)
42.0
```

Because case discrimination on an either type is so common, there is even (in Haskell) a standard higher-order function by which the same conversion can be expressed more concisely:

```
asFloat :: IntOrFloat -> Float
asFloat = either fromIntegral id
```

Specific either types can also be expressed by other means than a designated type constructor for such types. For instance, in functional programming with algebraic data types, a specific type can be declared for a given sum. For instance, the sum over *Int* and *Float* could also be declared like this:

```
data IntOrFloat = Int Int | Float Float
```

(We reuse type names as constructor symbols here, which is possible in Haskell, as these are separate namespaces.) The earlier conversion function is now to be defined by ordinary case discrimination over a (non-polymorphic) algebraic data type:

```
asFloat :: IntOrFloat -> Float
asFloat (Int x) = fromIntegral x
asFloat (Float x) = x
```

The advantage of the either type constructor is that it captures universally (polymorphically) the notion of disjoint (labeled) sum. Clearly, sums with more than two cases can be expressed by nested applications of the type constructor.

Metadata

- http://hackage.haskell.org/package/base-4.2.0.1/docs/Data-Either.html
- Vocabulary:Haskell