Script: Monads

Headline

Lecture "Monads" as part of Course:Lambdas in Koblenz

Description

Applications of pure functions return the same result whenever provided with the same arguments; they do not have any side effects. This may be viewed as a limitation when we need to model more general computations in functional programming. However, there is a functional programming abstraction, the monad, which comes to rescue. A monad is essentially an abstract data type to facilitate the composition of computations as opposed to functions. There are various monads to deal with all the computations effects that one may encounter, e.g., the State monad, the Maybe monad, the Reader monad, the Writer monad, and the IO monad. In modern Haskell, monads also interact with and relate to applicative functor, which may, in fact, also provide an alternative to the use of monads in some cases, but we limit our discussion to monads here for brevity.

Concepts

- Monad
- State monad
- Maybe monad
- Writer monad

Languages

- Language:Haskell

Features

- Feature:Logging

Contributions

- Contribution:haskellLogging
- Contribution:haskellWriter

Metadata

- Course:Lambdas in Koblenz
- Script:Functors_and_friends

- Script:Unparsing_and_parsing_in_Haskell

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: Monad

Headline

A functional programming idiom for computing effects

Illustration

The term "monad" originates from category theory, but this illustration focuses on the functional programming view where "monad" refers to a programming idiom for composing computations, specifically computations that may involve side effects or I/O actions. Monads have been popularized by Language:Haskell.

In Haskell, monads are developed and used with the help of thetype class *Monad* which is parametrized by a type constructor for the actual monad. Here is a sketch of the type class:

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  -- ... some details omitted
```

The *return* function serves the construction of trivial computations, i.e., computations that return values. The *>>=* (also knows as the bind function) compose a computation with a function that consumes the value of said computation to produce a composed computation. Here are some informal descriptions of popular monads:

- State monad
  - *return* v: return value *v* and pass on state
  - *bind* c f: apply computation *c* as state transformer and pass on transformed state to *f*
- Reader monad
  - *return* v: return value *v* and ignore environment
  - *bind* c f: pass environment to both *c* and *f*
- Writer monad
  - *return* v: return value *v* and empty output
  - *bind* c f: compose output from both *c* and *f*
- Maybe monad
  - *return* v: return "successful" value *v*
  - *bind* c f: fail if *c* fails, otherwise, pass on successful result to *f*

Metadata

- Vocabulary:Functional programming
- Type class
- Programming idiom
- http://www.haskell.org/haskellwiki/Monad

- http://en.wikipedia.org/wiki/Monad_%28functional_programming%29
- http://en.wikipedia.org/wiki/Monad_(category_theory)
- http://en.wikibooks.org/wiki/Haskell/Understanding_monads

Concept: Writer monad

Headline

A monad for synthesizing results or output

Illustration

Let us put to work the writer monad in a simple interpreter.

A baseline interpreter

There are these expression forms:

```
-- Simple Boolean expressions
data Expr = Constant Bool | And Expr Expr | Or Expr Expr
  deriving (Eq, Show, Read)
```

For instance, the following expression should evaluate to true:

```
-- A sample term with two operations
sample = And (Constant True) (Or (Constant False) (Constant True))
```

Here is a simple interpreter, indeed:

```
-- A straightforward interpreter
eval :: Expr -> Bool
eval (Constant b) = b
eval (And e1 e2) = eval e1 && eval e2
eval (Or e1 e2) = eval e1 || eval e2
```

Adding counting to the interpreter

Now suppose that the interpreter should also return the number of operations applied. We count *And* and *Or*as operations. Thus, the sample term should count as 2. We may incorporate counting into the initial interpreter as follows:

```
-- Interpreter with counting operations
eval' :: Expr -> (Bool, Int)
eval' (Constant b) = (b, 0)
eval' (And e1 e2) =
  let
    (b1,i) = eval' e1
    (b2,i') = eval' e2
  in (b1 && b2, i+i'+1)
eval' (Or e1 e2) =
  let
    (b1,i) = eval' e1
    (b2,i') = eval' e2
  in (b1 || b2, i+i'+1)
```

Alas, the resulting interpreter is harder to understand. The collection of counts is entangled with the basic logic.

Monadic style

By conversion to monadic style, we can hide counting except when we need increment the counter. We use the Writer monad here so that we simply combine counts from subexpression (as also done in the non-monadic code above). We could also be using the state monad, if we wanted to really track the operations counter along evaluation; this would be useful if we were adding an expression form for retrieving the count.

```
evalM :: Expr -> Writer (Sum Int) Bool
evalM (Constant b) = return b
evalM (And e1 e2) =
  evalM e1 >>= \b1 ->
  evalM e2 >>= \b2 ->
  tell (Sum 1) >>
  return (b1 && b2)
evalM (Or e1 e2) =
  evalM e1 >>= \b1 ->
  evalM e2 >>= \b2 ->
  tell (Sum 1) >>
  return (b1 || b2)
```

We can also use do notation:

```
-- Monadic style interpreter in do notation
evalM :: Expr -> Writer (Sum Int) Bool
evalM (Constant b) = return b
evalM (And e1 e2) = do
  b1 <- evalM e1
  b2 <- evalM e2
  tell (Sum 1)
  return (b1 && b2)
evalM (Or e1 e2) = do
  b1 <- evalM e1
  b2 <- evalM e2
  tell (Sum 1)
  return (b1 || b2)
```

The Writer monad

The Writer monad is readily provided by the Haskell library (in Control.Monad.Trans.Writer.Lazy), but we may want to understand how it might have been implemented. The data type for the Writer monad could look like this:

```
-- Computations as pairs of value and "output"
newtype Writer w a = Writer { runWriter :: (a, w) }
```

Thus, a stateful computation is basically a function a value with some output. The output type is assumed to be monoid because an empty output and the combination of outputs is uniformly defined in this manner.

The corresponding instance of the type class *Monad* follows:

```
-- Monad instance for Writer
instance Monoid w => Monad (Writer w)
  where
    return a = Writer (a, mempty)
    (Writer (a, w)) >>= f =
      let (Writer (b, w')) = f a in
        (Writer (b, w `mappend` w'))
```

The definition of *return* conveys that a pure computation produces the empty output. The definition of *bind* conveys that outputs are to be combined (in a certain order) from the operands of *bind*. Finally, we need to define the writer-specific operation *tell* for producing ouput:

```
-- Produce output
tell :: w -> Writer w ()
tell w = Writer ((), w)
```

In modern Haskell, we also need to make *Writer* an instance of *Applicative* (for applicative functors and *Functor* (for functors). This code is omitted here, but see GitHub for this page.

See Contribution:haskellWriter for a contribution which uses the writer monad.

Metadata

- http://en.wikipedia.org/wiki/Monad (functional programming)#Writer monad
- http://dl.acm.org/citation.cfm?id=647698.734150
- http://books.google.com/books?id=miO8bwAACAAJ
- Monad
- Vocabulary:Functional programming

Concept: State monad

Headline

A monad for state

Illustration

Let us put to work the State monad in a simple interpreter.

A baseline interpreter

There are these expression forms:

```
-- Simple Boolean expressions
data Expr = Constant Bool | And Expr Expr | Or Expr Expr
  deriving (Eq, Show, Read)
```

For instance, the following expression should evaluate to true:

```
sample = And (Constant True) (Or (Constant False) (Constant True))
```

Here is a simple interpreter, indeed:

```
-- A straightforward interpreter
eval :: Expr -> Bool
eval (Constant b) = b
eval (And e1 e2) = eval e1 && eval e2
eval (Or e1 e2) = eval e1 || eval e2
```

Adding counting to the interpreter

Now suppose that the interpreter should keep track of the number of operations applied. We count *And* and *Or*as operations. Thus, the sample term should count as 2. We may incorporate counting into the initial interpreter by essentially passing state for the counter. (We could also synthesize the count as output; see the illustration of the writer monad.) Thus:

```
-- Interpreter with counting operations
eval' :: Expr -> Int -> (Bool, Int)
eval' (Constant b) i = (b, i)
eval' (And e1 e2) i =
  let
    (b1,i') = eval' e1 i
    (b2,i'') = eval' e2 i'
  in (b1 && b2, i''+1)
eval' (Or e1 e2) i =
  let
    (b1,i') = eval' e1 i
    (b2,i'') = eval' e2 i'
  in (b1 || b2, i''+1)
```

Alas, the resulting interpreter is harder to understand. The threading of counts is entangled with the basic logic.

Monadic style

By conversion to monadic style, we can hide counting except when we need increment the counter. We use the State monad here so that we really track the operations counter along evaluation; this would be useful if we were adding an expression form for retrieving the count. We could also be using the writer monad, if we were only interested in the final count.

```
-- Monadic style interpreter
evalM :: Expr -> State Int Bool
evalM (Constant b) = return b
evalM (And e1 e2) =
  evalM e1 >>= \b1 ->
  evalM e2 >>= \b2 ->
  modify (+1) >>
  return (b1 && b2)
evalM (Or e1 e2) =
  evalM e1 >>= \b1 ->
  evalM e2 >>= \b2 ->
  modify (+1) >>
  return (b1 || b2)
```

We can also use do notation:

```
-- Monadic style interpreter in do notation
evalM' :: Expr -> State Int Bool
evalM' (Constant b) = return b
evalM' (And e1 e2) = do
  b1 <- evalM' e1
  b2 <- evalM' e2
  modify (+1)
  return (b1 && b2)
evalM' (Or e1 e2) = do
  b1 <- evalM' e1
  b2 <- evalM' e2
  modify (+1)
  return (b1 || b2)
```

The State monad

The state monad is readily provided by the Haskell library (in Control.Monad.State.Lazy), but we may want to understand how it might have been implemented. The data type for the State monad could look like this:

```
-- Data type for the State monad
newtype State s a = State { runState :: s -> (a,s) }
```

Thus, a stateful computation is basically a function on state which also returns a value.

The corresponding instance of the type class *Monad* follows:

```
-- Monad instance for State
instance Monad (State s)
  where
```

```
  return x = State (\s -> (x, s))
  c >>= f = State (\s -> let (x,s') = runState c s in runState (f x) s')
```

The definition of *return* conveys that a pure computation preserves the state. The definition of *bind* conveys that the state is to be threaded from the first argument to the second. Finally, we need to define the state-specific operation *modify* for accessing state:

```
-- Modification of state
modify :: (s -> s) -> State s ()
modify f = State (\s -> ((), f s))
```

In modern Haskell, we also need to make*State* an instance of *Applicative* (for applicative functors and *Functor* (for functors). This code is omitted here, but see GitHub for this page.

Metadata

- Monad
- http://www.haskell.org/haskellwiki/State_Monad
- http://cvs.haskell.org/Hugs/pages/libraries/mtl/Control-Monad-State.html

---

Contribution: haskellLogging

Headline

Logging in Haskell with non-monadic code

Characteristics

Starting from a straightforward family of functions for cutting salaries, the concern of logging the salary changes is incorporated into the functions such that the function results are enriched by the log entries for salary changes. This code is relatively verbose and implies poor abstraction. In particular, functionality for composing logs is scattered all over the functions. Ultimately, such a problem must be addressed with monads.

Illustration

Salary changes can be tracked in logs as follows:

```
type Log = [LogEntry]

data LogEntry =
    LogEntry {
      name :: String,
      oldSalary :: Float,
      newSalary :: Float
    }
  deriving (Show)
```

Here are a few entries resulting from a salary cut for the sample company:

```
[LogEntry {name = "Craig", oldSalary = 123456.0, newSalary = 61728.0},
 LogEntry {name = "Erik", oldSalary = 12345.0, newSalary = 6172.5},
 LogEntry {name = "Ralf", oldSalary = 1234.0, newSalary = 617.0},
 LogEntry {name = "Ray", oldSalary = 234567.0, newSalary = 117283.5},
 LogEntry {name = "Klaus", oldSalary = 23456.0, newSalary = 11728.0},
 LogEntry {name = "Karl", oldSalary = 2345.0, newSalary = 1172.5},
 LogEntry {name = "Joe", oldSalary = 2344.0, newSalary = 1172.0}]
```

Given a log, the median of salary deltas can be computed as follows:

```
log2median :: Log -> Float
log2median = median . log2deltas

log2deltas :: Log -> [Float]
log2deltas = sort . map delta
  where
    delta entry = newSalary entry - oldSalary entry
```

The above log reduces to the following median:

-6172.5

Feature:Cut is implemented in logging-enabled fashion as follows:

```
cut :: Company -> (Company, Log)
cut (Company n ds) = (Company n ds', log)
  where
   (ds', logs) = unzip (map cutD ds)
   log = concat logs
   cutD :: Department -> (Department, Log)
   cutD (Department n m ds es)
     = (Department n m' ds' es', log)
     where
       (m',log1) = cutE m
       (ds', logs2) = unzip (map cutD ds)
       (es', logs3) = unzip (map cutE es)
       log = concat ([log1]++logs2++logs3)
       cutE :: Employee -> (Employee, Log)
       cutE (Employee n a s) = (e', log)
         where
           e' = Employee n a s'
           s' = s/2
           log = [ LogEntry {
                    name = n,
                    oldSalary = s,
                    newSalary = s'
                 } ]
```

Thus, all functions return a regular data item (i.e., some part of the company) and a corresponding log. When lists of company parts are processed with map, then the lists of results must be unzipped (to go from a list of pairs to a pair of lists). In the function for departments, multiple logs arise for parts a department; these intermediate logs must be composed.

Relationships

- See Contribution:haskellComposition for the corresponding contribution that does not yet involve logging. The data model is preserved in the present contribution, but the functions for cutting salaries had to be rewritten since the logging concern crosscuts the function.
- See Contribution:haskellWriter for a variation on the present contribution, which leverages a writer monad, though, for conciseness and proper abstraction.

Architecture

There are these Haskell modules:

- Company.hs: the data model reused from Contribution:haskellComposition.
- Cut.hs: the combined implementation of Feature:Cut and Feature:Logging.
- Log.hs: types and functions for logs of salary changes needed for Feature:Logging.
- Main.hs: demonstration of all functions.

The contribution relies on the hackage package hstats.

Metadata

- Language:Haskell
- Language:Haskell 98
- Technology:GHC
- Technology:Cabal
- Feature:Hierarchical company
- Feature:Cut
- Feature:Logging
- Feature:Closed serialization
- Contributor:rlaemmel
- Theme:Haskell introduction

Contribution: haskellWriter

Headline

Logging in Haskell with the Writer monad

Characteristics

Salary changes are logged in aLanguage:Haskell-based implementation with the help of awriter monad. Compared to a non-monadic implementation, the code is more concise. Details of logging are localized such that they only surface in the context of code that actually changes salaries.

Illustration

See Contribution:haskellLogging for a simpler, non-monadic implementation.

The present, monadic implementation differs only with regard to the cut function:

```
cut :: Company -> Writer Log Company
cut (Company n ds) =
  do
    ds' <- mapM cutD ds
    return (Company n ds')
  where
    cutD :: Department -> Writer Log Department
    cutD (Department n m ds es) =
      do
        m' <- cutE m
        ds' <- mapM cutD ds
        es' <- mapM cutE es
        return (Department n m' ds' es')
      where
        cutE :: Employee -> Writer Log Employee
        cutE (Employee n a s) =
          do
            let s' = s/2
            let log = [ LogEntry {
                          name = n,
                          oldSalary = s,
                          newSalary = s'
                        } ]
            tell log
            return (Employee n a s')
```

Thus, the family of functions uses awriter monad in the result types. The sub-traversals are all composed by monadic bind (possibly expressed in do-notation). The function for processing departments totally abstracts from the fact that logging is involved. In fact, that function could be defined to be parametrically polymorphic in the monad at hand.

Relationships

- See Contribution:haskellComposition for the corresponding contribution that does not yet involve logging. The data model is preserved in the present contribution, but the functions for cutting salaries had to be rewritten since the logging concern crosscuts the function.
- See Contribution:haskellLogging for a variation on the present contribution which does not yet use monadic style.

Architecture

See Contribution:haskellLogging.

Metadata

- Language:Haskell
- Technology:GHC
- Technology:Cabal
- Technology:HUnit
- Writer monad
- Feature:Hierarchical company
- Feature:Cut
- Feature:Logging
- Contributor:tschmorleiz
- Contributor:rlaemmel
- Theme:Haskell potpourri
- Theme:Haskell introduction

Feature: Logging

Headline

Log and analyze salary changes

Description

Salaries of employees may change over time. For instance, a salary cut systematically decreases salaries. Of course, a pay raise could also happen; point-wise salary changes are conceivable as well. Salary changes are to be logged so that they can be analyzed within some window of interest. Specifically, a salary cut is to be logged with names of affected employees, salary before the change, and salary after the change. The log is to be analyzed in a statistical manner to determine the median and the mean of all salary deltas.

Motivation

The feature requires logging of updates to employee salaries. Depending on the programming language at hand, such logging may necessitate revision of the code that changes salaries. Specifically, logging of salary changes according to a salary cut may necessitate adaptation of the actual transformation for cutting salaries. Logging should be preferably added to a system while obeying separation of concerns. So logging is potentially a crosscutting concern, which may end being implemented in a scattered manner, unless some strong means of modularization can be adopted.

Illustration

The log for salary cut for the "standard" sample company would look as follows.

```
[ LogEntry {name = "Craig", oldSalary = 123456.0, newSalary = 61728.0},
  LogEntry {name = "Erik", oldSalary = 12345.0, newSalary = 6172.5},
  LogEntry {name = "Ralf", oldSalary = 1234.0, newSalary = 617.0},
  LogEntry {name = "Ray", oldSalary = 234567.0, newSalary = 117283.5},
  LogEntry {name = "Klaus", oldSalary = 23456.0, newSalary = 11728.0},
  LogEntry {name = "Karl", oldSalary = 2345.0, newSalary = 1172.5},
  LogEntry {name = "Joe", oldSalary = 2344.0, newSalary = 1172.0}
]
```

For what it matters, the salary cut operates as a depth-first, left-to-right traversal of the company; thus the order of the entries in the log. Projection of changes to deltas and sorting them results in the following list of deltas:

```
[ -117283.5,
  -61728.0,
  -11728.0,
  -6172.5,
  -1172.5,
```

```
  -1172.0,
  -617.0
]
```

Clearly, the median is the element in the middle:

```
-6172.5
```

By contrast, the mean is much different because of the skewed distribution of salaries:

```
-28553.355
```

See Contribution:haskellLogging for a simple implementation of the feature inLanguage:Haskell.

Relationships

- The present feature builds on top of Feature:Cut, as it is required to demonstrate the analysis of logged deltas for the transformation of a 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 the type for logs should involve the term "log".
- A suitable *demonstration* of the feature's implementation should cut the sample company and compute the median of the salary deltas, as indeed stipulated above.

Metadata

- Functional requirement
- Optional feature
- Separation of concerns

Concept: Maybe monad

Headline

A monad for dealing with partiality or error handling

Illustration

Let us put to work the Maybe monad in a simple interpreter.

A baseline interpreter

There are these expression forms for floats, addition, and square roots:

```
-- Simple arithmetic expressions
data Expr = Constant Float | Add Expr Expr | Sqrt Expr
  deriving (Eq, Show, Read)
```

Consider these samples:

```
-- Sample terms
sample = Sqrt (Constant 4)
sample' = Sqrt (Constant (-1))
```

The first expression should evaluate to 2.0. Evaluation should somehow fail for the second one. The most straightforward interpreter may be this one:

```
-- A straightforward interpreter
eval :: Expr -> Float
eval (Constant f) = f
eval (Add e1 e2) = eval e1 + eval e2
eval (Sqrt e) = sqrt (eval e)
```

Adding error handling to the interpreter

This interpreter would return *NaN* (not a number) for the second sample. This is suboptimal if we want to represent the error situation explicitly as an error value so that we cannot possibly miss the problem and it is propagated properly. To this end, we may use a Maybe type in the interpreter as follows:

```
-- An interpreter using a Maybe type for partiality
eval' :: Expr -> Maybe Float
eval' (Constant f) = Just f
eval' (Add e1 e2) =
  case eval' e1 of
    Nothing -> Nothing
    Just f1 ->
      case eval' e2 of
        Nothing -> Nothing
        Just f2 -> Just (f1 + f2)
```

```
eval' (Sqrt e) =
  case eval' e of
    Nothing -> Nothing
    Just f -> if f < 0.0
              then Nothing
              else Just (sqrt f)
```

Alas, the resulting interpreter is harder to understand. Maybes need to be handled for all subexpressions and the intention of propagating *Nothing* is expressed time and again.

Monadic style

By conversion to monadic style, we can hide error handling:

```
-- A monadic style interpreter
evalM :: Expr -> Maybe Float
evalM (Constant f) = return f
evalM (Add e1 e2) =
  evalM e1 >>= \f1 ->
  evalM e2 >>= \f2 ->
  return (f1 + f2)
evalM (Sqrt e) =
  evalM e >>= \f ->
  guard (f >= 0.0) >>
  return (sqrt f)
```

We can also use do notation:

```
-- A monadic style interpreter in do notation
evalM' :: Expr -> Maybe Float
evalM' (Constant f) = return f
evalM' (Add e1 e2) = do
  f1 <- evalM' e1
  f2 <- evalM' e2
  return (f1 + f2)
evalM' (Sqrt e) = do
  f <- evalM' e
  guard (f >= 0.0)
  return (sqrt f)
```

The Maybe monad

The Maybe monad is readily provided by the Haskell library, but we may want to understand how it might have been implemented. The corresponding instance of the type class *Monad* follows:

```
-- Monad instance for Maybe
instance Monad Maybe
  where
    return = Just
    Nothing >>= f = Nothing
    (Just x) >>= f = f x
```

The definition of *return* conveys that a pure computation is successful. The definition of *bind* conveys that *Nothing* for the first argument is to be propagated. The Maybe monad actually is a more special monad, i.e., a monad with *+* and *0*:

```
-- Type class MonadPlus (see Control.Monad)
class Monad m => MonadPlus m
  where
    mzero :: m a
    mplus :: m a -> m a -> m a

-- MonadPlus instance for Maybe
instance MonadPlus Maybe
  where
    mzero = Nothing
    Nothing `mplus` y = y
    x `mplus` _ = x
```

The Haskell library provides the *guard* function, which we used in the interpreter:

```
-- Succeed or fail
guard :: MonadPlus m => Bool -> m ()
guard b = if b then return () else mzero
```

In modern Haskell, we also need to make *Maybe* an instance of *Applicative* (for applicative functors and *Functor* (for functors). This code is omitted here, but see GitHub for this page.

Metadata

- Monad

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: Applicative functor

Headline

A functor with function application within the functor

Description

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

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

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

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

The following laws are assumed.

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

Illustration

We make *Maybe* and lists applicative functors:

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

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

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

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

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

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

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

Consider the following application.

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

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

Metadata

- Programming idiom
- Vocabulary:Functional programming
- http://www.haskell.org/haskellwiki/Applicative_functor
- https://en.wikibooks.org/wiki/Haskell/Applicative_functors
- http://dx.doi.org/10.1017/S0956796807006326
- http://learnyouahaskell.com/functors-applicative-functors-and-monoids

---

Concept: Reader monad

Headline

A monad for environment passing

Metadata

- Monad
- http://monads.haskell.cz/html/readermonad.html
