Часть Ⅱ

26.10.2021

Оглавление

[ЛИЦЕНЗИОННОЕ УВЕДОМЛЕНИЕ 2](#_Toc86438205)

[АППЛИКАТИВНЫЕ ФУНКТОРЫ 3](#_Toc86438206)

[Определение аппликативного функтора 3](#_Toc86438207)

[Представители класса типов Applicative 6](#_Toc86438208)

[Аппликативный парсер Parsec 13](#_Toc86438209)

[Аппликативный парсер своими руками 15](#_Toc86438210)

[Композиция на уровне типов 19](#_Toc86438211)

[УПРАВЛЕНИЕ ЭФФЕКТАМИ 22](#_Toc86438212)

[Класс типов Foldable 22](#_Toc86438213)

[Класс типов Traversable 26](#_Toc86438214)

[Законы и свойства класса Traversable 30](#_Toc86438215)

[Связь классов Monad и Applicative 33](#_Toc86438216)

[Классы типов Alternative и MonadPlus 36](#_Toc86438217)

[МОНАДЫ И ЭФФЕКТЫ 43](#_Toc86438218)

[Монада Except 43](#_Toc86438219)

[Монада Cont 48](#_Toc86438220)

[Трансформеры монад 53](#_Toc86438221)

[Трансформер ReaderT 59](#_Toc86438222)

[ТРАНСФОРМЕРЫ МОНАД 64](#_Toc86438223)

[Трансформер WriterT 64](#_Toc86438224)

[Трансформер StateT 68](#_Toc86438225)

[Трансформер ExceptT 71](#_Toc86438226)

[Неявный лифтинг 77](#_Toc86438227)

[Задачи на трансформеры 83](#_Toc86438228)

[ГРАФИК АКТИВНОСТИ 92](#_Toc86438229)

[СЕРТИФИКАТ 93](#_Toc86438230)

# ЛИЦЕНЗИОННОЕ УВЕДОМЛЕНИЕ

Настоящий отчет включает в себя оригинальные тексты задач курса «Функциональное программирование на языке Haskell (часть 2)», доступный по адресу <https://stepik.org/course/693>.

# АППЛИКАТИВНЫЕ ФУНКТОРЫ

## Определение аппликативного функтора

### Шаг 4:

В модуле Data.Functor определен оператор <$>, являющийся инфиксным аналогом функции fmap:

GHCi> :info <$>

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

-- Defined in `Data.Functor'

infixl 4 <$>

В выражении succ <$> "abc" этот оператор имеет тип (Char -> Char) -> [Char] -> [Char]. Какой тип имеет первое (левое) вхождение этого оператора в выражении succ <$> succ <$> "abc"?

Ответ:

(Char -> Char) -> (Char -> Char) -> (Char -> Char)

### Шаг 5:

Сделайте типы данных Arr2 e1 e2 и Arr3 e1 e2 e3 представителями класса типов Functor:

newtype Arr2 e1 e2 a = Arr2 { getArr2 :: e1 -> e2 -> a }

newtype Arr3 e1 e2 e3 a = Arr3 { getArr3 :: e1 -> e2 -> e3 -> a }

Эти типы инкапсулируют вычисление с двумя и тремя независимыми окружениями соответственно:

GHCi> getArr2 (fmap length (Arr2 take)) 10 "abc"

3

GHCi> getArr3 (tail <$> tail <$> Arr3 zipWith) (+) [1,2,3,4] [10,20,30,40,50]

[33,44]

Ответ:

newtype Arr2 e1 e2 a = Arr2 {getArr2 :: e1 -> e2 -> a}

newtype Arr3 e1 e2 e3 a = Arr3 {getArr3 :: e1 -> e2 -> e3 -> a}

instance Functor (Arr2 e1 e2) where

fmap f = Arr2 . ((fmap.fmap.fmap) f getArr2)

instance Functor (Arr3 e1 e2 e3) where

fmap f = Arr3 . ((fmap.fmap.fmap.fmap) f getArr3)

{-

Поэтапный переход от лямбды до бесточечного стиля

fmap f arr = Arr2 (\a b-> f $ getArr2 arr a b)

fmap f arr = Arr2 (\a b-> (f.getArr2 arr a) b)

fmap f arr = Arr2 (\a b-> ((.) f (getArr2 arr a)) b)

fmap f arr = Arr2 (\a b-> ((f.) (getArr2 arr a)) b) -- Важнейший шаг

fmap f arr = Arr2 (\a -> ( ((f.).(getArr2 arr)) a ))

fmap f arr = Arr2 ( (f.).(getArr2 arr) )

fmap f arr = Arr2 $ ((f.).) (getArr2 arr)

fmap f arr = Arr2 $ (fmap (fmap f)) (getArr2 arr)

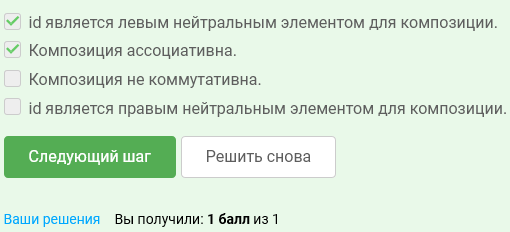
fmap f arr = Arr2 $ (fmap.fmap) f (getArr2 arr)

-}

### Шаг 7:

Самостоятельно докажите выполнение первого (fmap id = id) и второго (fmap f . fmap g = fmap (f . g)) законов функторов для функтора частично примененной функциональной стрелки (->) e. Отметьте те свойства оператора композиции функций, которыми вы воспользовались.

Ответ:



### Шаг 9:

Докажите выполнение второго закона функторов для функтора списка: fmap f (fmap g xs) = fmap (f . g) xs. Предполагается, что все списки конечны и не содержат расходимостей.

Ответ:

|  |
| --- |
| **Доказательство выполнения второго закона функтора для списка**  В соответствие с условий задания, требуется доказать или опровергнуть справедливость следующий Гипотезы: fmap f (fmap g xs) ≡ fmap (f . g) xs, где xs - конечный список а f и g функции, последовательно примененные к элементам этого списка, гарантируется отсутствие расходимостей.  Далее, эксплуатируя исходные коды стандартной библиотеки компилятора GHC введем реализацию функции fmap для списков. Просмотр [источника](https://hackage.haskell.org/package/base-4.16.0.0/docs/src/GHC.Base.html#line-1158) позволяет прямым образом констатировать, что вызов fmap выполняет делегирование до функции map, в которой находиться определяющая реализация. Логика исполнения функции map отражена в том же модуле но в другом [местоположении](https://hackage.haskell.org/package/base-4.16.0.0/docs/src/GHC.Base.html#map).  Руководствуясь перечисленными источниками, сформулируем воплощении функции fmap но с легким упрощением - обращение к функции map исключено путем непосредственного встраивания:  fmap \_ []     = [] --Правило (1)  fmap f (x:xs) = f x : fmap f xs --Правило (2)  Помимо этого, для нужд нижеизложенного доказательства, строки пронумерованы, что позволит выполнять ссылки на них.  Непосредственное доказательство гипотезы осуществимо методом структурной индукцией. Настоящий метод предусматривает две фазы: База индукции и Индукционный переход. Первая фаза посвящена обоснованию актуальности тривиального случая, во второй фаза, предполагается верность текущего случая и доказывается достоверность следующего(ближайшего) случая.  **База индукции**  База индукции нацелена на доказательство "простейшего" варианта, беря в аналогию эффект домино, это такой вариант, движение от которого возможен только в одном направление. Поэтому, база индукции, в данном случае, выступает пустой список.   1. fmap f (fmap g []) ≡ fmap (f . g) [] - Подстановка пустого списка в определение Гипотезы. 2. fmap f [] ≡ [] - Употребление правила (1) для левой и правой части 3. [] ≡ [] Повторное употребление правила (1)   **Вывод:** Пункт 3 привел к тождеству, что говорит об истинности Гипотезы для пустых списков.  **Индукционный переход**  В индукционным переходе предполагается истинность гипотезы для списка из n-элементов и проводится исследование поведения этой гипотезы для списка, к которому добавлен один элемент:   1. fmap f (fmap g (x:xs)) ≡ fmap (f . g) (x:xs) - Подстановка списка x:xs в определение Гипотезы. 2. fmap f (g x : fmap g xs) ≡ (f . g) x : fmap (f . g) xs - Употребление правила (2) 3. (f $ g x) : fmap f (fmap g xs) ≡ (f . g) x : fmap (f . g) xs - Повторное употребление правила (2) 4. (f . g) x : fmap f (fmap g xs) ≡ (f . g) x : fmap (f . g) xs - Преобразование операторов аппликации и применения к композиции (левая часть).   **Вывод:** В соответствие с индукционным предположением, гипотеза верна для списка xs, однако, имеет место быть эквивалентность для списков x:xs, которая обусловлена тем, что левая и правая часть подверглись одинаковым преобразованием - списки левой и правой части пополнились элементом (f . g) x. Выполнение второго закона функторов для списков доказано. |

### Шаг 15:

Следующий тип данных задает гомогенную тройку элементов, которую можно рассматривать как трехмерный вектор:

data Triple a = Tr a a a deriving (Eq,Show)

Сделайте этот тип функтором и аппликативным функтором с естественной для векторов семантикой покоординатного применения:

GHCi> (^2) <$> Tr 1 (-2) 3

Tr 1 4 9

GHCi> Tr (^2) (+2) (\*3) <\*> Tr 2 3 4

Tr 4 5 12

Ответ:

data Triple a = Tr a a a deriving (Eq,Show)

instance Functor Triple where

fmap = (<\*>).pure

instance Applicative Triple where

pure x = Tr x x x

(<\*>) (Tr f1 f2 f3) (Tr x y z) = Tr (f1 x) (f2 y) (f3 z)

## Представители класса типов Applicative

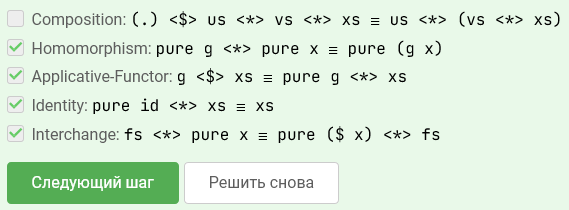
### Шаг 3:

Предположим, что для стандартного функтора списка оператор (<\*>) определен стандартным образом, а метод pure изменен на

pure x = [x,x]

К каким законам класса типов Applicative будут в этом случае существовать контрпримеры?

Ответ:



### Шаг 5:

В модуле Data.List имеется семейство функций zipWith, zipWith3, zipWith4,..:

GHCi> let x1s = [1,2,3]

GHCi> let x2s = [4,5,6]

GHCi> let x3s = [7,8,9]

GHCi> let x4s = [10,11,12]

GHCi> zipWith (\a b -> 2\*a+3\*b) x1s x2s

[14,19,24]

GHCi> zipWith3 (\a b c -> 2\*a+3\*b+5\*c) x1s x2s x3s

[49,59,69]

GHCi> zipWith4 (\a b c d -> 2\*a+3\*b+5\*c-4\*d) x1s x2s x3s x4s

[9,15,21]

Аппликативные функторы могут заменить всё это семейство

GHCi> getZipList $ (\a b -> 2\*a+3\*b) <$> ZipList x1s <\*> ZipList x2s

[14,19,24]

GHCi> getZipList $ (\a b c -> 2\*a+3\*b+5\*c) <$> ZipList x1s <\*> ZipList x2s <\*> ZipList x3s

[49,59,69]

GHCi> getZipList $ (\a b c d -> 2\*a+3\*b+5\*c-4\*d) <$> ZipList x1s <\*> ZipList x2s <\*>ZipList x3s <\*> ZipList x4s

[9,15,21]

Реализуйте операторы (>\*<) и (>$<), позволяющие спрятать упаковку ZipList и распаковку getZipList:

GHCi> (\a b -> 2\*a+3\*b) >$< x1s >\*< x2s

[14,19,24]

GHCi> (\a b c -> 2\*a+3\*b+5\*c) >$< x1s >\*< x2s >\*< x3s

[49,59,69]

GHCi> (\a b c d -> 2\*a+3\*b+5\*c-4\*d) >$< x1s >\*< x2s >\*< x3s >\*< x4s

[9,15,21]

Ответ:

import Control.Applicative(ZipList(ZipList), getZipList)

(>$<) = (<$>)

fs >\*< vs = getZipList $ ZipList fs <\*> ZipList vs

infixl 4 >$<, >\*<

### Шаг 8:

Функция

divideList :: Fractional a => [a] -> a

divideList [] = 1

divideList (x:xs) = (/) x (divideList xs)

сворачивает список посредством деления. Модифицируйте ее, реализовав divideList' :: (Show a, Fractional a) => [a] -> (String,a), такую что последовательность вычислений отражается в логе:

GHCi> divideList [3,4,5]

3.75

GHCi> divideList' [3,4,5]

("<-3.0/<-4.0/<-5.0/1.0",3.75)

Используйте аппликативный функтор пары, сохраняя близкую к исходной функции структуру реализации

divideList' :: (Show a, Fractional a) => [a] -> (String,a)

divideList' [] = \_

divideList' (x:xs) = (/) <$> \_ <\*> \_

Ответ:

divideList' :: (Show a, Fractional a) => [a] -> (String,a)

divideList' [] = ("1.0",1)

divideList' (x:xs) = (/) <$> ("<-"++show x++"/", x) <\*> divideList' xs

### Шаг 10:

Сделайте типы данных Arr2 e1 e2 и Arr3 e1 e2 e3 представителями класса типов Applicative

newtype Arr2 e1 e2 a = Arr2 { getArr2 :: e1 -> e2 -> a }

newtype Arr3 e1 e2 e3 a = Arr3 { getArr3 :: e1 -> e2 -> e3 -> a }

с естественной семантикой двух и трех окружений:

GHCi> getArr2 (Arr2 (\x y z -> x+y-z) <\*> Arr2 (\*)) 2 3

-1

GHCi> getArr3 (Arr3 (\x y z w -> x+y+z-w) <\*> Arr3 (\x y z -> x\*y\*z)) 2 3 4

-15

Ответ:

newtype Arr2 e1 e2 a = Arr2 { getArr2 :: e1 -> e2 -> a }

newtype Arr3 e1 e2 e3 a = Arr3 { getArr3 :: e1 -> e2 -> e3 -> a }

instance Functor (Arr2 e1 e2) where

fmap = (<\*>) . pure

instance Functor (Arr3 e1 e2 e3) where

fmap = (<\*>) . pure

instance Applicative (Arr2 e1 e2) where

pure = Arr2 . const.const

(<\*>) (Arr2 f) (Arr2 c) = Arr2 (\e1 e2 -> f e1 e2 (c e1 e2))

instance Applicative (Arr3 e1 e2 e3) where

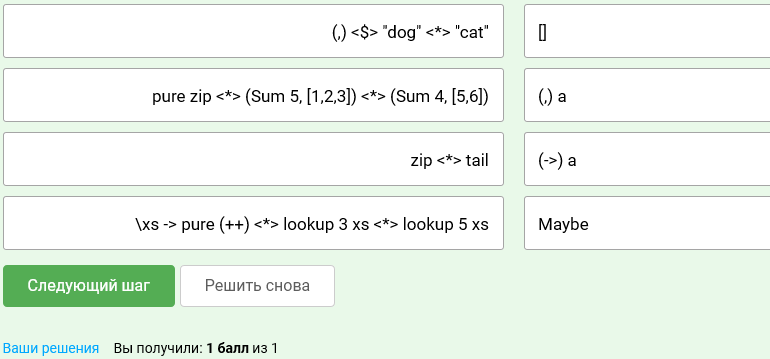
pure = Arr3 . const.const.const

(<\*>) (Arr3 f) (Arr3 c) = Arr3 (\e1 e2 e3 -> f e1 e2 e3 (c e1 e2 e3) )

### Шаг 11:

Сопоставьте вычислению, поднятому в аппликативный функтор, конкретного представителя класса типов Applicative, в котором это вычисление происходит.

Ответ:



### Шаг 14:

Двойственный оператор аппликации (<\*\*>) из модуля Control.Applicative изменяет направление вычислений, не меняя порядок эффектов:

infixl 4 <\*\*>

(<\*\*>) :: Applicative f => f a -> f (a -> b) -> f b

(<\*\*>) = liftA2 (flip ($))

Определим оператор (<\*?>) с той же сигнатурой, что и у (<\*\*>), но другой реализацией:

infixl 4 <\*?>

(<\*?>) :: Applicative f => f a -> f (a -> b) -> f b

(<\*?>) = flip (<\*>)

Для каких стандартных представителей класса типов Applicative можно привести цепочку аппликативных вычислений, дающую разный результат в зависимости от того, какой из этих операторов использовался?

В следующих шести примерах вашей задачей будет привести такие контрпримеры для стандартных типов данных, для которых они существуют. Следует заменить аппликативное выражение в предложении in на выражение того же типа, однако дающее разные результаты при вызовах с (<??>) = (<\*\*>) и (<??>) = (<\*?>). Проверки имеют вид exprXXX (<\*\*>) == exprXXX (<\*?>) для различных имеющихся XXX. Если вы считаете, что контрпримера не существует, то менять ничего не надо.

Ответ:

{-# LANGUAGE RankNTypes#-}

import Control.Applicative ((<\*\*>),ZipList(..))

infixl 4 <\*?>

(<\*?>) :: Applicative f => f a -> f (a -> b) -> f b

(<\*?>) = flip (<\*>)

exprMaybe :: (forall a b . Maybe a -> Maybe (a -> b) -> Maybe b) -> Maybe Int

exprMaybe op =

let (<??>) = op

infixl 4 <??>

in Just 5 <??> Just (+2) -- place for counterexample

exprList :: (forall a b . [a] -> [a -> b] -> [b]) -> [Int]

exprList op =

let (<??>) = op

infixl 4 <??>

in [0,1] <??> [(1-), (1+)]

exprZipList :: (forall a b . ZipList a -> ZipList (a -> b) -> ZipList b) -> ZipList Int

exprZipList op =

let (<??>) = op

infixl 4 <??>

in ZipList [1,2] <??> ZipList [(+3),(+4)] -- place for counterexample

exprEither :: (forall a b . Either String a -> Either String (a -> b) -> Either String b) -> Either String Int

exprEither op =

let (<??>) = op

infixl 4 <??>

in Left "a" <??> Left "b"

exprPair :: (forall a b . (String,a) -> (String,a -> b) -> (String,b)) -> (String,Int)

exprPair op =

let (<??>) = op

infixl 4 <??>

in ("a", 0) <??> ("b", id)

exprEnv :: (forall a b . (String -> a) -> (String -> (a -> b)) -> (String -> b)) -> (String -> Int)

exprEnv op =

let (<??>) = op

infixl 4 <??>

in length <??> (\\_ -> (+5)) -- place for counterexample

{-

Выполним бызовый переход(Б.П.) для оператора <\*?>

v <\*?> f ==> f <\*> v

Выполним бызовый переход для оператора <\*\*>

v <\*\*> f ==>

(<\*\*>) v f ==>

liftA2 (flip ($)) v f ==>

flip (\f x -> f x) <$> v <\*> f ==>

(\a b-> b a) <$> v <\*> f ==>

-}

{-

Доказательство для списков

-----------------------

[a,b,c] <\*?> [f1,f2,f3]

-----------------------

После Б.П.

[f1,f2,f3] <\*> [a,b,c] ==>

[f1 a, f1 b, f1 c,

f2 a, f2 b, f2 c,

f3 a, f3 b, f3 c]

-----------------------

[a,b,c] <\*\*> [f1,f2,f3]

-----------------------

После Б.П.

(\a b-> b a) <$> [a,b,c] <\*> [f1,f2,f3] ==>

[\q->q a,\q->q b,\q->q c] <\*> [f1,f2,f3] ==>

[\q->q a,\q->q b,\q->q c] <\*> [f1,f2,f3] ==>

[ f1 a, f2 a, f3 a,

f1 b, f2 b, f3 b,

f1 c, f2 c, f3 c]

Вывод: Результативные списки не равны, гипотеза ложна

Контрпример: [0,1] vs [(1-), (1+)]

-}

{-

Доказательство для списков ZipList

Прим: Обусловимся, что списки не в контейнерах, но аппликатив этого контейнера используется как основной

-----------------------

[a,b,c] <\*?> [f1,f2,f3]

-----------------------

После Б.П.

[f1,f2,f3] <\*> [a,b,c] ==>

[f1 a, f2 b, f3 c]

-----------------------

[a,b,c] <\*\*> [f1,f2,f3]

-----------------------

После Б.П.

(\a b-> b a) <$> [a,b,c] <\*> [f1,f2,f3] ==>

[\q->q a,\q->q b,\q->q c] <\*> [f1,f2,f3] ==>

[\q->q a,\q->q b,\q->q c] <\*> [f1,f2,f3] ==>

[f1 a, f2 b, f3 c]

Вывод: Результативные списки равны, гипотеза подтверждена

-}

{-

Доказательство для пар

-----------------------

(m1,v) <\*?> (m2,f)

-----------------------

После Б.П.

(m2,f) <\*> (m1,v) ==>

(m2 `mappend` m1, f v)

-----------------------

(m1,v) <\*\*> (m2,f)

-----------------------

После Б.П.

(\a b-> b a) <$> (m1,v) <\*> (m2,f)

(m1,(\q-> q v)) <\*> (m2,f)

(m1 `mappend` m2,f v)

Вывод: Результативные пары не равны, гипотеза ложна

Контрпример: ("a", 0) vs ("b",id)

-}

{-

Доказательство для Maybe

-----------------------

Just v <\*?> Just f

-----------------------

После Б.П.

Just f <\*> Just v

Just (f v)

-----------------------

Just v <\*\*> Just f

-----------------------

После Б.П.

(\a b-> b a) <$> Just v <\*> Just f

Just (\b-> b v) <\*> Just f

Just (f v)

Вывод: Для контейнера Maybe и конструктора Just гипотеза подтверждена

Прим: Имперически следует, что наличие Nothing приведет к Nothing в

любом случаи

-}

{-

Доказательство для Either

-----------------------

Left v <\*?> Left f

-----------------------

После Б.П.

Left f <\*> Left v

f

-----------------------

Left v <\*\*> Left f

-----------------------

После Б.П.

(\a b-> b a) <$> Left v <\*> Left f

v

Вывод: Для контейнера Either гипотеза провалена

Контрпример: Left "a" vs Left "b"

-}

{-

Доказательство для стрелочных типов

-----------------------

e->v <\*?> (e->v->w)

-----------------------

После Б.П.

f1:= (e->v->w) <\*> f2:= e->v ==>

\e->f1 e (f2 e)

-----------------------

(e->v) <\*\*> (e->v->w)

-----------------------

После Б.П.

(\a b-> b a) <$> (e->v) <\*> (e->v->w)

\b e -> b (f2 e) <\*> (e->v->w)

\e->f1 e (f2 e)

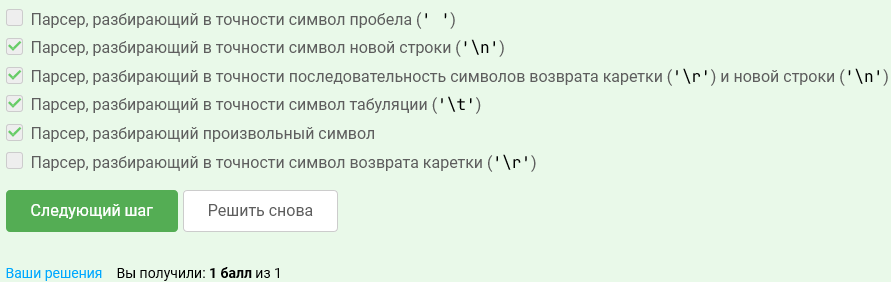
-}

## Аппликативный парсер Parsec

### Шаг 3:

Какие из следующих примитивных парсеров имеются в библиотеке Text.Parsec.Char ?

Ответ:



### Шаг 5:

Реализуйте парсер getList, который разбирает строки из чисел, разделенных точкой с запятой, и возвращает список строк, представляющих собой эти числа:

GHCi> parseTest getList "1;234;56"

["1","234","56"]

GHCi> parseTest getList "1;234;56;"

parse error at (line 1, column 10):

unexpected end of input

expecting digit

GHCi> parseTest getList "1;;234;56"

parse error at (line 1, column 3):

unexpected ";"

expecting digit

Совет: изучите парсер-комбинаторы, доступные в модуле Text.Parsec, и постарайтесь найти наиболее компактное решение.

Ответ:

import Text.Parsec

getList :: Parsec String u [String]

getList = number `sepBy` char ';'

where

number = many1 digit

### Шаг 7:

Используя аппликативный интерфейс Parsec, реализуйте функцию ignoreBraces, которая принимает три аргумента-парсера. Первый парсер разбирает текст, интерпретируемый как открывающая скобка, второй — как закрывающая, а третий разбирает весь входной поток, расположенный между этими скобками. Возвращаемый парсер возвращает результат работы третьего парсера, скобки игнорируются.

GHCi> test = ignoreBraces (string "[[") (string "]]") (many1 letter)

GHCi> parseTest test "[[ABC]]DEF"

"ABC"

Ответ:

import Text.Parsec

ignoreBraces :: Parsec [Char] u a -> Parsec [Char] u b -> Parsec [Char] u c -> Parsec [Char] u c

ignoreBraces o c p = o\*>p<\*c

## Аппликативный парсер своими руками

### Шаг 4:

Предположим, тип парсера определен следующим образом:

newtype Prs a = Prs { runPrs :: String -> Maybe (a, String) }

Сделайте этот парсер представителем класса типов Functor. Реализуйте также парсер anyChr :: Prs Char, удачно разбирающий и возвращающий любой первый символ любой непустой входной строки.

GHCi> runPrs anyChr "ABC"

Just ('A',"BC")

GHCi> runPrs anyChr ""

Nothing

GHCi> runPrs (digitToInt <$> anyChr) "BCD"

Just (11,"CD")

Ответ:

instance Functor Prs where

fmap f p = let

f' text = (\(v,t)->(f v,t)) <$> runPrs p text

in Prs f'

anyChr :: Prs Char

anyChr = let

f (s:ls) = Just (s,ls)

f \_ = Nothing

in Prs f

### Шаг 6:

Сделайте парсер

newtype Prs a = Prs { runPrs :: String -> Maybe (a, String) }

из предыдущей задачи аппликативным функтором с естественной для парсера семантикой:

GHCi> runPrs ((,,) <$> anyChr <\*> anyChr <\*> anyChr) "ABCDE"

Just (('A','B','C'),"DE")

GHCi> runPrs (anyChr \*> anyChr) "ABCDE"

Just ('B',"CDE")

Представитель для класса типов Functor уже реализован.

Ответ:

instance Applicative Prs where

pure x = Prs $ \s-> Just (x,s)

{-

<\*> :: (c (v->w))-> c v-> c w

let c = String -> Maybe (a, String)

(String -> Maybe (v->w, String))-> (String -> Maybe (v, String))-> (String -> Maybe (w, String))

-}

(Prs pf) <\*> (Prs p) = Prs (\s-> do

(f,text)<-pf s

(v,text')<-p text

return (f v,text'))

### Шаг 8:

Рассмотрим более продвинутый парсер, позволяющий возвращать пользователю причину неудачи при синтаксическом разборе:

newtype PrsE a = PrsE { runPrsE :: String -> Either String (a, String) }

Реализуйте функцию satisfyE :: (Char -> Bool) -> PrsE Char таким образом, чтобы функция

charE :: Char -> PrsE Char

charE c = satisfyE (== c)

обладала бы следующим поведением:

GHCi> runPrsE (charE 'A') "ABC"

Right ('A',"BC")

GHCi> runPrsE (charE 'A') "BCD"

Left "unexpected B"

GHCi> runPrsE (charE 'A') ""

Left "unexpected end of input"

Ответ:

satisfyE :: (Char -> Bool) -> PrsE Char

satisfyE cf = PrsE f where

f (s:ls) | cf s = Right (s,ls)

| otherwise = Left $ "unexpected " ++ [s]

f \_ = Left "unexpected end of input"

charE :: Char -> PrsE Char

charE c = satisfyE (== c)

### Шаг 9:

Сделайте парсер

newtype PrsE a = PrsE { runPrsE :: String -> Either String (a, String) }

из предыдущей задачи функтором и аппликативным функтором:

GHCi> let anyE = satisfyE (const True)

GHCi> runPrsE ((,) <$> anyE <\* charE 'B' <\*> anyE) "ABCDE"

Right (('A','C'),"DE")

GHCi> runPrsE ((,) <$> anyE <\* charE 'C' <\*> anyE) "ABCDE"

Left "unexpected B"

GHCi> runPrsE ((,) <$> anyE <\* charE 'B' <\*> anyE) "AB"

Left "unexpected end of input"

Ответ:

instance Functor PrsE where

fmap pf p = let

f text = (\(v,t)->(pf v,t)) <$> runPrsE p text

in PrsE f

instance Applicative PrsE where

pure x = PrsE $ \s-> Right (x,s)

(PrsE pf) <\*> (PrsE p) = PrsE (\s-> do

(f,text)<-pf s

(v,text')<-p text

return (f v,text'))

### Шаг 12:

Сделайте парсер

newtype Prs a = Prs { runPrs :: String -> Maybe (a, String) }

представителем класса типов Alternative с естественной для парсера семантикой:

GHCi> runPrs (char 'A' <|> char 'B') "ABC"

Just ('A',"BC")

GHCi> runPrs (char 'A' <|> char 'B') "BCD"

Just ('B',"CD")

GHCi> runPrs (char 'A' <|> char 'B') "CDE"

Nothing

Представители для классов типов Functor и Applicative уже реализованы. Функцию char :: Char -> Prs Char включать в решение не нужно, но полезно реализовать для локального тестирования.

Ответ:

--import Control.Applicative(Alternative(..))

instance Alternative Prs where

empty = Prs $ const Nothing

(<|>) (Prs p) (Prs p') = Prs $ (<|>) . p <\*> p'

{-

1)Prs $ \s-> p s <|> p' s

2)Prs $ (\s->(<|>) (p s)) <\*> p'

3)Prs $ (<|>) . p <\*> p'

-}

### Шаг 14:

Реализуйте для парсера

newtype Prs a = Prs { runPrs :: String -> Maybe (a, String) }

парсер-комбинатор many1 :: Prs a -> Prs [a], который отличается от many только тем, что он терпит неудачу в случае, когда парсер-аргумент неудачен на начале входной строки.

> runPrs (many1 $ char 'A') "AAABCDE"

Just ("AAA","BCDE")

> runPrs (many1 $ char 'A') "BCDE"

Nothing

Функцию char :: Char -> Prs Char включать в решение не нужно, но полезно реализовать для локального тестирования.

Ответ:

{-

many1 p = let peek = (:) <$> p in

peek <\*> many1 p <|> peek <\*> pure []

Accept distributivity

peek <\*> (many1 p <|> pure [])

-}

many1 :: Prs a -> Prs [a]

many1 p = (:) <$> p <\*> (many1 p <|> pure [])

### Шаг 15:

Реализуйте парсер nat :: Prs Int для натуральных чисел, так чтобы парсер

mult :: Prs Int

mult = (\*) <$> nat <\* char '\*' <\*> nat

обладал таким поведением

GHCi> runPrs mult "14\*3"

Just (42,"")

GHCi> runPrs mult "64\*32"

Just (2048,"")

GHCi> runPrs mult "77\*0"

Just (0,"")

GHCi> runPrs mult "2\*77AAA"

Just (154,"AAA")

Реализацию функции char :: Char -> Prs Char следует включить в присылаемое решение, только если она нужна для реализации парсера nat.

Ответ:

import Data.Char(isDigit)

satisfy :: (Char -> Bool) -> Prs Char

satisfy cf = Prs (\txt-> case txt of

(s:ls)->if cf s then Just(s,ls) else Nothing

\_->Nothing)

nat :: Prs Int

nat = read <$> some (satisfy isDigit)

## Композиция на уровне типов

### Шаг 3:

Населите допустимыми нерасходящимися выражениями следующие типы

type A = ((,) Integer |.| (,) Char) Bool

type B t = ((,,) Bool (t -> t) |.| Either String) Int

type C = (|.|) ((->) Bool) ((->) Integer) Integer

Ответ:

type A = ((,) Integer |.| (,) Char) Bool

{-

((Integer,) |.| (,) Char) Bool

((Integer,) |.| (Char,) ) Bool

(|.|) (Integer,(Char,)) Bool

(|.|) (Integer,(Char,Bool))

-}

type B t = ((,,) Bool (t -> t) |.| Either String) Int

{-

((,,) Bool (t -> t) |.| Either String) Int

(|.|) (Bool,(t -> t),Either String) Int

(|.|) (Bool,(t -> t),Either String Int)

-}

type C = (|.|) ((->) Bool) ((->) Integer) Integer

{-

(|.|) ((->) Bool) ((->) Integer) Integer

(|.|) (Bool->) (Integer->) Integer

(|.|) (Bool->Integer->Integer)

-}

a :: A

a = Cmps (8, ('#', True))

b :: B t

b = Cmps (False, id, Left ">>=")

c :: C

c = Cmps $ flip const

### Шаг 5:

Сделайте тип

newtype Cmps3 f g h a = Cmps3 { getCmps3 :: f (g (h a)) }

deriving (Eq,Show)

представителем класса типов Functor при условии, что первые его три параметра являются функторами:

GHCi> fmap (^2) $ Cmps3 [[[1],[2,3,4],[5,6]],[],[[7,8],[9,10,11]]]

Cmps3 {getCmps3 = [[[1],[4,9,16],[25,36]],[],[[49,64],[81,100,121]]]}

Ответ:

newtype Cmps3 f g h a = Cmps3 { getCmps3 :: f (g (h a)) }

deriving (Eq,Show)

infixl 1 &

x & f = f x

instance (Functor f, Functor g, Functor h)=>Functor (Cmps3 f g h) where

{-

1) fmap f (Cmps3 v) = Cmps3 $ fmap (\q->fmap (\w->fmap (\e->f e) w) q) v

2) fmap f (Cmps3 v) = Cmps3 $ fmap (fmap (fmap(f))) v

3) fmap f (Cmps3 v) = Cmps3 $ (fmap f & fmap & fmap) v

4) fmap f = Cmps3 . (fmap f & fmap & fmap).getCmps3

-}

fmap f = Cmps3 . (fmap f & fmap & fmap) . getCmps3

### Шаг 7:

Докажите выполнение второго закона функторов для композиции двух функторов:

fmap h2 (fmap h1 (Cmps x)) = fmap (h2 . h1) (Cmps x).

Ответ:

|  |
| --- |
| В качестве подготовительного шага введем несколько правил и свойств, которые необходимы для корректного доказательства:   1. fmap f (Cmps x) = Cmps $ fmap (fmap f) x - Типовая реализация функции fmap интерфейса Functor для композиции типов (Правило №1) 2. f (g a) ≡ (f . g) a - Определение композиции (Правило №2) 3. fmap f . fmap g ≡ fmap (f . g) - Второй закон функтора (Правило №3)   Далее будет произведено фактическое доказательство целевой гипотезы при помощи цепи эквивалентных преобразований. Для успешности осуществления этого процесса предполагается, что типы участвующие в формулировки гипотезы являются представителями интерфейса Functor, более того, второй закон, описанный в правиле №3, должен быть истинен для этих типов и отсутствие расходимостей обязательно.   1. fmap h2 (fmap h1 (Cmps x)) ≡ fmap (h2 . h1) (Cmps x) - Формулировка гипотезы, данная по условию задания. 2. fmap h2 (Cmps $ fmap (fmap h1) x) ≡ Cmps $ fmap (fmap (h2 . h1)) x - Применение тела реализации функции fmap для композиции типов (Правила №1) к внутренним подвыражениям левой и правой части. 3. Cmps $ fmap (fmap h2) (fmap (fmap h1) x) ≡ Cmps $ fmap (fmap (h2 . h1)) x - Повторное применение Правила №1 к левой части. 4. Cmps $ (fmap (fmap h2) . fmap (fmap h1)) x ≡ Cmps $ fmap (fmap (h2 . h1)) x - Задействование определения композиции (Правило №2) для левой части. 5. Cmps $ fmap ((fmap h2) . (fmap h1)) x ≡ Cmps $ fmap (fmap (h2 . h1)) x - Использование второго закона функтора (Правило №3) для левой части. 6. Cmps $ fmap (fmap (h2 . h1)) x ≡ Cmps $ fmap (fmap (h2 . h1)) x - Повторное использование второго закона функтора (Правило №3) для левой части.   **Вывод:** Шаг 6 привел к непосредственному тождеству, следовательно, гипотеза о справедливости второго закона функторов для композиции типов **подтверждена** |

### Шаг 9:

Напишите универсальные функции

unCmps3 :: Functor f => (f |.| g |.| h) a -> f (g (h a))

unCmps4 :: (Functor f2, Functor f1) => (f2 |.| f1 |.| g |.| h) a -> f2 (f1 (g (h a)))

позволяющие избавляться от синтаксического шума для композиции нескольких функторов:

GHCi> pure 42 :: ([] |.| [] |.| []) Int

Cmps {getCmps = [Cmps {getCmps = [[42]]}]}

GHCi> unCmps3 (pure 42 :: ([] |.| [] |.| []) Int)

[[[42]]]

GHCi> unCmps3 (pure 42 :: ([] |.| Maybe |.| []) Int)

[Just [42]]

GHCi> unCmps4 (pure 42 :: ([] |.| [] |.| [] |.| []) Int)

[[[[42]]]]

Ответ:

unCmps3 :: Functor f => (f |.| g |.| h) a -> f (g (h a))

unCmps3 = fmap getCmps . getCmps

unCmps4 :: (Functor f2, Functor f1) => (f2 |.| f1 |.| g |.| h) a -> f2 (f1 (g (h a)))

unCmps4 = fmap unCmps3 . getCmps

# УПРАВЛЕНИЕ ЭФФЕКТАМИ

## Класс типов Foldable

### Шаг 4:

Сделайте тип

data Triple a = Tr a a a deriving (Eq,Show)

представителем класса типов Foldable:

GHCi> foldr (++) "!!" (Tr "ab" "cd" "efg")

"abcdefg!!"

GHCi> foldl (++) "!!" (Tr "ab" "cd" "efg")

"!!abcdefg"

Ответ:

import Data.Foldable(Foldable)

--data Triple a = Tr a a a deriving (Eq,Show)

instance Foldable Triple where

--foldr :: (a->b->b) -> b -> (c a) -> b

--foldl :: (b->a->b) -> b -> (c a) -> b

foldr f ini (Tr a b c) = a `f` (b `f` (c `f` ini)) --f a $ f b $ f c ini

foldl f ini (Tr a b c) = ((ini `f` a) `f` b) `f` c --f (f (f ini a) b) c

### Шаг 6:

Для реализации свертки двоичных деревьев нужно выбрать алгоритм обхода узлов дерева (см., например, <http://en.wikipedia.org/wiki/Tree_traversal>).

Сделайте двоичное дерево

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

представителем класса типов Foldable, реализовав симметричную стратегию (in-order traversal). Реализуйте также три другие стандартные стратегии (pre-order traversal, post-order traversal и level-order traversal), сделав типы-обертки

newtype Preorder a = PreO (Tree a) deriving (Eq, Show)

newtype Postorder a = PostO (Tree a) deriving (Eq, Show)

newtype Levelorder a = LevelO (Tree a) deriving (Eq, Show)

представителями класса Foldable.

GHCi> tree = Branch (Branch Nil 1 (Branch Nil 2 Nil)) 3 (Branch Nil 4 Nil)

GHCi> foldr (:) [] tree

[1,2,3,4]

GHCi> foldr (:) [] $ PreO tree

[3,1,2,4]

GHCi> foldr (:) [] $ PostO tree

[2,1,4,3]

GHCi> foldr (:) [] $ LevelO tree

[3,1,4,2]

Ответ:

{-import Data.Foldable(Foldable)

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

newtype Preorder a = PreO (Tree a) deriving (Eq, Show)

newtype Postorder a = PostO (Tree a) deriving (Eq, Show)

newtype Levelorder a = LevelO (Tree a) deriving (Eq, Show)-}

instance Foldable Tree where

foldr f ini (Branch l v r) = foldr f (f v $ foldr f ini r) l --((\x->foldr f x l) . f v . (\x->foldr f x r)) ini

foldr f ini Nil = ini

instance Foldable Preorder where

foldr f ini (PreO tr) = fld ini tr where

fld ini (Branch l v r) = f v $ fld (fld ini r) l --(f v . (\x->fld x l) . (\x->fld x r)) ini

fld ini Nil = ini

instance Foldable Postorder where

foldr f ini (PostO tr) = fld ini tr where

fld ini (Branch l v r) = fld (fld (f v ini) r) l --((\x->fld x l) . (\x->fld x r) . f v ) ini

fld ini Nil = ini

instance Foldable Levelorder where

foldr f ini (LevelO tr) = fld ini [tr] where

fld ini (Branch l v r :xs) = f v $ fld ini $ xs ++ [l, r]

fld ini (Nil :xs) = fld ini xs

fld ini [] = ini

### Шаг 8:

Предположим, что определены следующие функции

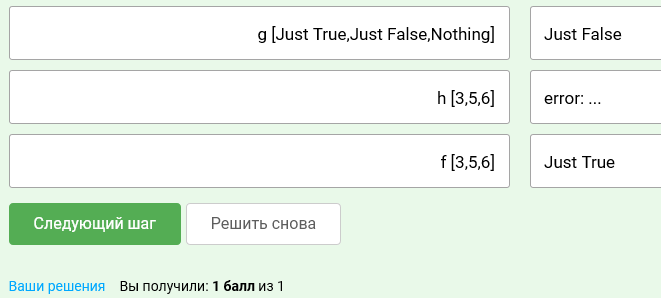
f = Just . getAny . foldMap Any . fmap even

g = getLast . foldMap Last

h = Just . getAll . foldMap All . map isDigit

Сопоставьте их вызовы и результаты этих вызовов. Предполагается, что загружены все модули, требующиеся для доступа к использованным функциям и конструкторам данных.

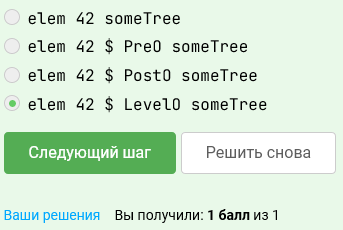
Ответ:



### Шаг 10:

Предположим, что у нас реализованы все свертки, основанные на разных стратегиях обхода дерева из предыдущей задачи. Какой из вызовов «лучше определен», то есть возвращает результат на более широком классе деревьев?

Ответ:



### Шаг 13:

Реализуйте функцию

mkEndo :: Foldable t => t (a -> a) -> Endo a

принимающую контейнер функций и последовательно сцепляющую элементы этого контейнера с помощью композиции, порождая в итоге эндоморфизм.

GHCi> e1 = mkEndo [(+5),(\*3),(^2)]

GHCi> appEndo e1 2

17

GHCi> e2 = mkEndo (42,(\*3))

GHCi> appEndo e2 2

6

Ответ:

import Data.Foldable(Foldable)

import Data.Monoid(Endo(..))

mkEndo :: Foldable t => t (a -> a) -> Endo a

mkEndo = foldMap Endo --foldr (\s ls->Endo s<>ls) mempty

### Шаг 16:

Сделайте тип

infixr 9 |.|

newtype (|.|) f g a = Cmps { getCmps :: f (g a) } deriving (Eq,Show)

представителем класса типов Foldable при условии, что аргументы композиции являются представителями Foldable.

GHCi> maximum $ Cmps [Nothing, Just 2, Just 3]

3

GHCi> length $ Cmps [[1,2], [], [3,4,5,6,7]]

7

Ответ:

{-{-# LANGUAGE TypeOperators #-}

import Data.Monoid(Endo(..))

import Data.Foldable(Foldable)

infixr 9 |.|

newtype (|.|) f g a = Cmps { getCmps :: f (g a) } deriving (Eq,Show)-}

instance (Foldable f, Foldable g)=>Foldable (f |.| g) where

foldMap f = (foldMap . foldMap) f . getCmps

{-

foldMap :: Monoid m =>(a->m) ->t a->m

1) foldMap f (Cmps v) = foldMap (\q->foldMap (\w->f w) q) v

2) foldMap f (Cmps v) = foldMap (foldMap (\w->f w)) v

3) foldMap f (Cmps v) = foldMap (foldMap f) v

4) foldMap f (Cmps v) = foldMap (foldMap f) (getCmps v)

5) foldMap f = foldMap (foldMap f) . getCmps

-}

{-

foldr :: (a -> b -> b) -> b -> c a -> b

1) foldr f ini (Cmps c) = appEndo ( foldr (\q lq -> (foldr (\w lw-> Endo (f w) <>lw) mempty q) <>lq ) mempty c ) ini

2) foldr f ini (Cmps c) = appEndo ( foldr (\q lq -> (foldr (\w-> (<>) (Endo (f w))) mempty q) <>lq ) mempty c ) ini

3) foldr f ini (Cmps c) = appEndo ( foldr (\q lq -> (foldr ((<>) . Endo . f ) mempty q) <>lq ) mempty c ) ini

4) foldr f ini (Cmps c) = appEndo ( foldr (\q -> (<>) (foldr ((<>) . Endo . f ) mempty q) ) mempty c ) ini

5) foldr f ini (Cmps c) = appEndo (foldr ((<>) . (foldr ((<>) . Endo . f) mempty)) mempty c) ini

-}

## Класс типов Traversable

### Шаг 4:

Предположим для двоичного дерева

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

реализован представитель класса типов Foldable, обеспечивающий стратегию обхода pre-order traversal. Какую строку вернет следующий вызов

GHCi> tree = Branch (Branch Nil 1 Nil) 2 (Branch (Branch Nil 3 Nil) 4 (Branch Nil 5 Nil))

GHCi> fst $ sequenceA\_ $ (\x -> (show x,x)) <$> tree

Ответ:

"21435"

### Шаг 7:

Реализуйте функцию

traverse2list :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f [b]

работающую с эффектами как traverse\_,, но параллельно с накоплением эффектов «восстанавливающую» сворачиваемую структуру в виде списка:

GHCi> traverse2list (\x -> [x+10,x+20]) [1,2,3]

[[11,12,13],[11,12,23],[11,22,13],[11,22,23],[21,12,13],[21,12,23],[21,22,13],[21,22,23]]

GHCi> traverse2list (\x -> [x+10,x+20]) $ Branch (Branch Nil 1 Nil) 2 (Branch Nil 3 Nil)

[[12,11,13],[12,11,23],[12,21,13],[12,21,23],[22,11,13],[22,11,23],[22,21,13],[22,21,23]]

Ответ:

traverse2list :: (Foldable t, Applicative f) => (a -> f b) -> t a -> f [b]

traverse2list f = foldr (\x y -> pure (:) <\*> f x <\*> y ) $ pure []

### Шаг 11:

Сделайте тип

data Triple a = Tr a a a deriving (Eq,Show)

представителем класса типов Traversable:

GHCi> foldl (++) "!!" (Tr "ab" "cd" "efg")

"!!abcdefg"

GHCi> traverse (\x -> if x>10 then Right x else Left x) (Tr 12 14 16)

Right (Tr 12 14 16)

GHCi> traverse (\x -> if x>10 then Right x else Left x) (Tr 12 8 4)

Left 8

GHCi> sequenceA (Tr (Tr 1 2 3) (Tr 4 5 6) (Tr 7 8 9))

Tr (Tr 1 4 7) (Tr 2 5 8) (Tr 3 6 9)

Ответ:

{-data Triple a = Tr a a a deriving (Eq,Show)

instance Functor Triple where

fmap f (Tr x y z) = Tr (f x) (f y) (f z)

--https://stepik.org/lesson/30427/step/4?unit=11044

instance Foldable Triple where

foldr f ini (Tr a b c) = f a $ f b $ f c ini-}

instance Traversable Triple where

sequenceA (Tr x y z) = Tr<$>x<\*>y<\*>z

{-

Ход размышлений:

1) Известно, что левый опперанд <$> должен быть функцией, однако он также

может быть частично примененный функцией.

Выразим эту идею с помощью лямбда-абстракций:

((\x-> (\y-> Tr x y x))<$>x)<\*>y

Следовательно, путь к решению найден.

2) ((\x-> (\y z-> Tr x y z))<$>x)<\*>y<\*>z

3) ((\x y z-> Tr x y z)<$>x)<\*>y<\*>z

-}

### Шаг 12:

Сделайте тип данных

data Result a = Ok a | Error String deriving (Eq,Show)

представителем класса типов Traversable (и всех других необходимых классов типов).

GHCi> traverse (\x->[x+2,x-2]) (Ok 5)

[Ok 7,Ok 3]

GHCi> traverse (\x->[x+2,x-2]) (Error "!!!")

[Error "!!!"]

Ответ:

--data Result a = Ok a | Error String deriving (Eq,Show)

instance Functor Result where

fmap f (Ok v) = Ok $ f v

fmap \_ (Error s) = Error s

--Важная особенность: fmap \_ err@(Error s) = err

instance Foldable Result where

foldMap f (Ok v) = f v

foldMap \_ (Error s) = mempty

instance Traversable Result where

sequenceA (Ok v) = Ok <$> v

sequenceA (Error s) = pure $ Error s

### Шаг 14:

Сделайте двоичное дерево

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

представителем класса типов Traversable (а также всех других необходимых классов типов).

GHCi> traverse (\x -> if odd x then Right x else Left x) (Branch (Branch Nil 1 Nil) 3 Nil)

Right (Branch (Branch Nil 1 Nil) 3 Nil)

GHCi> traverse (\x -> if odd x then Right x else Left x) (Branch (Branch Nil 1 Nil) 2 Nil)

Left 2

GHCi> sequenceA $ Branch (Branch Nil [1,2] Nil) [3] Nil

[Branch (Branch Nil 1 Nil) 3 Nil,Branch (Branch Nil 2 Nil) 3 Nil]

Ответ:

import Data.Traversable(foldMapDefault, fmapDefault)

--data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

instance Foldable Tree where

foldMap = foldMapDefault

instance Functor Tree where

fmap = fmapDefault

instance Traversable Tree where

traverse f (Branch l v r) = flip Branch <$> f v <\*> traverse f l <\*> traverse f r -- Pre Order

--traverse f (Branch l v r) = Branch <$> traverse f l <\*> f v <\*> traverse f r --In Order

traverse \_ Nil = pure Nil

### Шаг 15:

Сделайте тип

infixr 9 |.|

newtype (|.|) f g a = Cmps { getCmps :: f (g a) } deriving (Eq,Show)

представителем класса типов Traversable при условии, что аргументы композиции являются представителями Traversable.

GHCi> sequenceA (Cmps [Just (Right 2), Nothing])

Right (Cmps {getCmps = [Just 2,Nothing]})

GHCi> sequenceA (Cmps [Just (Left 2), Nothing])

Left 2

Ответ:

{- {-# LANGUAGE TypeOperators #-}

infixr 9 |.|

newtype (|.|) f g a = Cmps { getCmps :: f (g a) } deriving (Eq,Show)

instance (Functor f, Functor g) => Functor (f |.| g) where

fmap f (Cmps v) = Cmps $ (fmap . fmap) f v

--fmap f = (Cmps . (fmap . fmap) f ) . getCmps

--fmap = ((. getCmps) . (Cmps .)) . (fmap . fmap)

instance (Foldable f, Foldable g) => Foldable (f |.| g) where

foldMap f = (foldMap . foldMap) f . getCmps-}

instance (Traversable f, Traversable g) => Traversable (f |.| g) where

traverse f (Cmps v) = Cmps <$> (traverse . traverse) f v

--c= Cmps [Just (Left 2), Nothing] :: (|.|) [] Maybe (Either Int ())

## Законы и свойства класса Traversable

### Шаг 5:

В предположении что обе части закона *composition* для sequenceA

sequenceA . fmap Compose == Compose . fmap sequenceA . sequenceA

имеют тип

(Applicative f, Applicative g, Traversable t) => t (f (g a)) -> Compose f g (t a)

укажите тип подвыражения fmap sequenceA в правой части. Контекст указывать не надо.

Ответ:  
f (t (g a)) -> f (g (t a))

### Шаг 6:

Рассмотрим следующий тип данных

data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

Этот тип представляет собой контейнер-последовательность, который по построению может содержать только нечетное число элементов:

GHCi> cnt1 = Un 42

GHCi> cnt3 = Bi 1 2 cnt1

GHCi> cnt5 = Bi 3 4 cnt3

GHCi> cnt5

Bi 3 4 (Bi 1 2 (Un 42))

GHCi> cntInf = Bi 'A' 'B' cntInf

GHCi> cntInf

Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'B' (Bi 'A' 'Interrupted.

GHCi>

Сделайте этот тип данных представителем классов типов Functor, Foldable и Traversable:

GHCi> (+1) <$> cnt5

Bi 4 5 (Bi 2 3 (Un 43))

GHCi> toList cnt5

[3,4,1,2,42]

GHCi> sum cnt5

52

GHCi> traverse (\x->[x+2,x-2]) cnt1

[Un 44,Un 40]

Ответ:

import Data.Traversable(fmapDefault, foldMapDefault)

--data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

instance Functor OddC where

fmap = fmapDefault

instance Foldable OddC where

foldMap = foldMapDefault

instance Traversable OddC where

traverse f (Bi v v' c) = Bi <$> f v <\*> f v' <\*> traverse f c --liftA3 Bi (f v) (f v') (traverse f c)

traverse f (Un v) = Un <$> f v

### Шаг 8:

Расширьте интерфейс для работы с температурами из предыдущего видео Кельвинами и реализуйте функцию

k2c :: Temperature Kelvin -> Temperature Celsius

обеспечивающую следующее поведение

GHCi> k2c 0

Temperature (-273.15)

GHCi> k2c 0 == Temperature (-273.15)

True

GHCi> k2c 273.15

Temperature 0.0

Ответ:

{-# LANGUAGE GeneralizedNewtypeDeriving #-}

newtype Temperature a = Temperature Double

deriving (Num,Show,Eq,Fractional)

data Celsius

data Fahrenheit

data Kelvin

comfortTemperature :: Temperature Celsius

comfortTemperature = Temperature 23

c2f :: Temperature Celsius -> Temperature Fahrenheit

c2f (Temperature c) = Temperature (1.8 \* c + 32)

k2c :: Temperature Kelvin -> Temperature Celsius

k2c (Temperature k) = Temperature $ k - 273.15

### Шаг 12:

Сделайте двоичное дерево

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

представителем класса типов Traversable таким образом, чтобы обеспечить для foldMapDefault порядок обхода «postorder traversal»:

GHCi> testTree = Branch (Branch (Branch Nil 1 Nil) 2 (Branch Nil 3 Nil)) 4 (Branch Nil 5 Nil)

GHCi> foldMapDefault (\x -> [x]) testTree

[1,3,2,5,4]

Ответ:

import Data.Traversable (foldMapDefault {-,fmapDefault-})

{-

data Tree a = Nil | Branch (Tree a) a (Tree a) deriving (Eq, Show)

instance Functor Tree where

fmap = fmapDefault

-}

instance Foldable Tree where

foldMap = foldMapDefault

instance Traversable Tree where

traverse f (Branch l v r) = flip . Branch <$> traverse f l <\*> traverse f r <\*> f v

traverse \_ Nil = pure Nil

--traverse f (Branch l v r) = liftA3 (flip . Branch) (traverse f l) (traverse f r) $ f v

{-

Функция fmapDefault не пригодна в качестве воплощения функтора(fmap),

если в классе Traversable функция traverse оставлена по умолчанию, реализация

по умолчанию для traverse ссылается на sequence. Наглядно:

1) fmap = fmapDefault ==>

fmap f = getConst . travere (Const . f)

Вызов traverse приводит к:

2) traverse f = sequenceA . fmap f

Вызов fmap порождает не завершающуюся рекурсию

-}

## Связь классов Monad и Applicative

### Шаг 7:

Сделайте парсер

newtype PrsE a = PrsE { runPrsE :: String -> Either String (a, String) }

из [первого модуля курса](https://stepik.org/lesson/30425/step/8-30425/step/8) представителем класса типов Monad:

GHCi> runPrsE (do {a <- charE 'A'; b <- charE 'B'; return (a,b)}) "ABC"

Right (('A','B'),"C")

GHCi> runPrsE (do {a <- charE 'A'; b <- charE 'B'; return (a,b)}) "ACD"

Left "unexpected C"

GHCi> runPrsE (do {a <- charE 'A'; b <- charE 'B'; return (a,b)}) "BCD"

Left "unexpected B"

Ответ:

instance Monad PrsE where

{-

(>>=) :: c a -> (a -> c b) -> c b

let c = Prs (String -> Either String (a, String)) then

PrsE (String -> Either String (a, String)) ->

(a -> PrsE (String -> Either String (b, String))) ->

PrsE (String -> Either String (b, String))

-}

(>>=) (PrsE p) f = PrsE $ (=<<) (uncurry $ runPrsE . f) . p

{-

1) PrsE (\s -> p s >>= (\(v,s')-> runPrsE (f v) s') )

2) PrsE (\s -> p s >>= (\q->uncurry (runPrsE . f) q) )

3) PrsE $ (=<<) (uncurry $ runPrsE . f) . p

-}

### Шаг 10:

Для типа данных

data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

(контейнер-последовательность, который по построению может содержать только нечетное число элементов) реализуйте функцию

concat3OC :: OddC a -> OddC a -> OddC a -> OddC a

конкатенирующую три таких контейнера в один:

GHCi> tst1 = Bi 'a' 'b' (Un 'c')

GHCi> tst2 = Bi 'd' 'e' (Bi 'f' 'g' (Un 'h'))

GHCi> tst3 = Bi 'i' 'j' (Un 'k')

GHCi> concat3OC tst1 tst2 tst3

Bi 'a' 'b' (Bi 'c' 'd' (Bi 'e' 'f' (Bi 'g' 'h' (Bi 'i' 'j' (Un 'k')))))

Обратите внимание, что соображения четности запрещают конкатенацию двух контейнеров OddC.

Реализуйте всё «честно», не сводя к стандартным спискам.

Ответ:

--data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

concat3OC :: OddC a -> OddC a -> OddC a -> OddC a

concat3OC c1 c2 c3 = let cnc v (Bi e e' t) = Bi v e $ cnc e' t

cnc v (Un e) = Bi v e c3

in case c1 of

Bi e e' t -> Bi e e' $ concat3OC t c2 c3

Un e -> cnc e c2

### Шаг 11:

Для типа данных

data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

реализуйте функцию

concatOC :: OddC (OddC a) -> OddC a

Она должна обеспечивать для типа OddC поведение, аналогичное поведению функции concat для списков:

GHCi> concatOC $ Un (Un 42)

Un 42

GHCi> tst1 = Bi 'a' 'b' (Un 'c')

GHCi> tst2 = Bi 'd' 'e' (Bi 'f' 'g' (Un 'h'))

GHCi> tst3 = Bi 'i' 'j' (Un 'k')

GHCi> concatOC $ Bi tst1 tst2 (Un tst3)

Bi 'a' 'b' (Bi 'c' 'd' (Bi 'e' 'f' (Bi 'g' 'h' (Bi 'i' 'j' (Un 'k')))))

Реализуйте всё «честно», не сводя к стандартным спискам.

Ответ:

--data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

concat3OC :: OddC a -> OddC a -> OddC a -> OddC a

concat3OC c1 c2 c3 = let cnc v (Bi l r t) = Bi v l $ cnc r t

cnc v (Un e) = Bi v e c3

in case c1 of

Bi l r t -> Bi l r $ concat3OC t c2 c3

Un e -> cnc e c2

concatOC :: OddC (OddC a) -> OddC a

concatOC (Bi l r t) = concat3OC l r $ concatOC t

concatOC (Un v) = v

--ctst1 = Bi (Bi 1 2 (Un 3)) (Bi 4 5 (Un 6)) (Bi (Bi 7 8 (Un 9)) (Bi 10 11 (Un 12)) (Un (Un 13)))

--print $ take 10 $ show(concatOC ( Bi inf inf (Un inf)))

### Шаг 12:

Сделайте тип данных

data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

представителем классов типов Functor, Applicative и Monad. Семантика должна быть подобной семантике представителей этих классов типов для списков: монада OddC должна иметь эффект вычисления с произвольным нечетным числом результатов:

GHCi> tst1 = Bi 10 20 (Un 30)

GHCi> tst2 = Bi 1 2 (Bi 3 4 (Un 5))

GHCi> do {x <- tst1; y <- tst2; return (x + y)}

Bi 11 12 (Bi 13 14 (Bi 15 21 (Bi 22 23 (Bi 24 25 (Bi 31 32 (Bi 33 34 (Un 35)))))))

GHCi> do {x <- tst2; y <- tst1; return (x + y)}

Bi 11 21 (Bi 31 12 (Bi 22 32 (Bi 13 23 (Bi 33 14 (Bi 24 34 (Bi 15 25 (Un 35)))))))

Функцию fail можно не реализовывать, полагаясь на реализацию по умолчанию.

Реализуйте всё «честно», не сводя к стандартным спискам.

Ответ:

--import Control.Monad(liftM, ap)

--data OddC a = Un a | Bi a a (OddC a) deriving (Eq,Show)

concat3OC :: OddC a -> OddC a -> OddC a -> OddC a

concat3OC c1 c2 c3 = let cnc e (Bi v v' t) = Bi e v $ cnc v' t

cnc e (Un v) = Bi e v c3

in case c1 of

Bi v v' t -> Bi v v' $ concat3OC t c2 c3

Un v -> cnc v c2

instance Functor OddC where

fmap = (=<<) . (return .) --liftM

instance Applicative OddC where

pure = return

f <\*> c = f >>= ((>>=) c . (return .)) --ap

instance Monad OddC where

return = Un

Bi v v' t >>= f = concat3OC (f v) (f v') $ t >>= f

Un v >>= f = f v

## Классы типов Alternative и MonadPlus

### Шаг 5:

Выполняются ли для стандартных представителей Applicative, Alternative, Monad и MonadPlus типа данных Maybe следующие законы дистрибутивности:

(u <|> v) <\*> w = u <\*> w <|> v <\*> w

(u `mplus` v) >>= k = (u >>= k) `mplus` (v >>= k)

Если нет, то приведите контрпример, если да, то доказательство.

Предполагается, что расходимости отсутствуют.

Ответ:

|  |
| --- |
| **Доказательство выполнения первого закона дистрибутивности для типа данных Maybe.**  Введём тройку обозначений, на которые будем ссылаться по тексту:   1. (u <|> v) <\*> w ≡ u <\*> w <|> v <\*> w - Гипотеза. 2. (u <|> v) <\*> w - Левая часть. 3. u <\*> w <|> v <\*> w - Правая часть.   Введение  Проверяемая гипотеза является формулировкой первого закона дистрибутивности, предположение о его выполнении должно быть подтверждено или опровергнуто этим доказательством, для типа Maybe. Тип Maybe имеет два конструктора данных Just и Nothing, дискретное рассмотрение каждого из них позволит лучшим образом структурировать излагаемый материал. В связи с этим работа включает две фазы: Первая фаза будет строго предполагать равенство u = Nothing, вторая фаза предназначена для анализа случая u = Just x.  Фаза 1 (u = Nothing)  Используя метод поэтапной свертки для Левой части, имеем:   1. (u <|> v) <\*> w 2. (Nothing <|> v) <\*> w (Замена u на значение) 3. v <\*> w (Так как Nothing <|> v ≡ v)   Используя метод поэтапной свертки для Правой части, имеем:   1. u <\*> w <|> v <\*> w 2. (u <\*> w) <|> (v <\*> w) (Оператор <\*> имеет больший приоритет) 3. (Nothing <\*> w) <|> (v <\*> w) 4. Nothing <|> (v <\*> w) (Так как Nothing <\*> w ≡ Nothing) 5. v <\*> w (Так как Nothing <|> v ≡ v)   Вывод: Тождественные преобразования Левой и Правой части привели к эквивалентным выражениям. Гипотеза истина при u = Nothing.  Фаза 2 (u = Just x) Используя метод поэтапной свертки для Левой части имеем:   1. (u <|> v) <\*> w 2. (Just x <|> v) <\*> w 3. Just x <\*> w (Так как Just x <|> v ≡ Just x)   Используя метод поэтапной свертки для Правой части, имеем:   1. u <\*> w <|> v <\*> w 2. (u <\*> w) <|> (v <\*> w) (Оператор <\*> имеет больший приоритет) 3. (Just x <\*> w) <|> (v <\*> w) 4. (Just $ fmap x w) <|> (v <\*> w) (Раскрытие оператора <\*>) 5. Just $ fmap x w (Так как Just a <|> b ≡ Just a) 6. Just x <\*> w   Вывод: Тождественные преобразования Левой и Правой части привели к эквивалентным выражениям. Гипотеза истина при u = Just x.  Заключение  Все конструкторы типа Maybe были рассмотрены в соответствующих фазах. В каждой фазе закон был справедлив, исходя из этого, актуальность Гипотезы: (u <|> v) <\*> w ≡ u <\*> w <|> v <\*> w подтверждена для любого случая, не содержащего расходимость. |
| **Опровержение второго закона дистрибутивности для типа данных Maybe.**  Для большей однозначности трактовки нижеописанного, присвоим имена основным объектам:   1. (u `mplus` v) >>= k ≡ (u >>= k) `mplus` (v >>= k) - Гипотеза (Второй закон дистрибутивности) 2. (u `mplus` v) >>= k - Левая часть 3. (u >>= k) `mplus` (v >>= k) - Правая часть   Введение  Настоящая работа предназначена для демонстрации последовательности рассуждений, в результате выполнение второго закона дистрибутивности (Гипотезы) для типа данных Maybe будет опровергнуто. Структура, по мимо введения, включает две фазы, служащие для почленного анализа каждого конструктора типа Maybe. Первая фаза руководствуется предположением о равенстве переменной u значению Nothing, во второй же фазе, делается противоположенное заявление - u = Just x.   Опровержение построено на эксплуатации метода поэтапной свертки, который применён к паре подвыражений Гипотезы.  Фаза 1 (u = Nothing)  Задействуем метод поэтапной свертки для Левой и Правой части, считая о справедливости равенства u = Notinhg.  Тогда левая часть Гипотезы примет вид:   1. (u `mplus` v) >>= k 2. (Nothing `mplus` v) >>= k 3. v >>= k   Проведём аналогичные действия для правой части:   1. (u `mplus` v) >>= k 2. (Nothing >>= k) `mplus` (v >>= k) 3. Nothing `mplus` (v >>= k) 4. v >>= k   Вывод: Преобразованные подвыражения Гипотезы эквиволентны, что делает её верной при u = Nothing.  Фаза 2 (u = Just x)  Использование пошаговой свертки при u = Just x для Левой части приведет к:   1. (u `mplus` v) >>= k 2. (Just x `mplus` v) >>= k 3. u >>= k   Нахождение уязвимости Правой части несколько сложнее и нуждается в более тонком анализе. По условию задачи, функция k не имеет ограничений, она произвольна, что делает возможным существование вырожденной k, возвращающая Nothing только для единственного значение, которое обозначено переменной x. Воспроизведем метод поэтапной свертки для представленного случая:   1. (u >>= k) `mplus` (v >>= k) 2. (Just x >>= k) `mplus` (v >>= k) 3. Nothing `mplus` (v >>= k) (Частный случай, k x = Nothing) 4. v >>= k (Так как Nothing `mplus` a = a)   Вывод: Анализ частного случая выявил сценарий, для которого равенство правой и левой части не выполнено. Как следствие, Гипотеза о истинности второго закона дистрибутивности для типа Maybe провалена.  **Контрпример:**  import Control.Monad(mplus) u = Just 5 v = Just 7 k x = if (x==5) then Nothing else Just x main = print $ ((u `mplus` v) >>= k,                 (u >>= k) `mplus` (v >>= k)) --Out: (Nothing, Just 7) |

### Шаг 6:

Предположим мы сделали парсер

newtype PrsE a = PrsE { runPrsE :: String -> Either String (a, String) }

представителем классов типов Alternative следующим образом

instance Alternative PrsE where

empty = PrsE f where

f \_ = Left "empty alternative"

p <|> q = PrsE f where

f s = let ps = runPrsE p s

in if null ps

then runPrsE q s

else ps

Эта реализация нарушает закон дистрибутивности для Alternative:

GHCi> runPrsE ((charE 'A' <|> charE 'B') \*> charE 'C') "ABC"

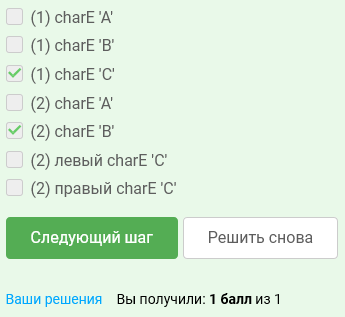
Left "unexpected B"

GHCi> runPrsE (charE 'A' \*> charE 'C' <|> charE 'B' \*> charE 'C') "ABC"

Left "unexpected A"

От какого парсера приходит сообщение об ошибке в первом и втором примерах?

Ответ:



### Шаг 7:

Реализуем улучшенную версию парсера PrsE

newtype PrsEP a = PrsEP { runPrsEP :: Int -> String -> (Int, Either String (a, String)) }

parseEP :: PrsEP a -> String -> Either String (a, String)

parseEP p = snd . runPrsEP p 0

Этот парсер получил дополнительный целочисленный параметр в аргументе и в возвращаемом значении. С помощью этого параметра мы сможем отслеживать и передвигать текущую позицию в разбираемой строке и сообщать о ней пользователю в случае ошибки:

GHCi> charEP c = satisfyEP (== c)

GHCi> runPrsEP (charEP 'A') 0 "ABC"

(1,Right ('A',"BC"))

> runPrsEP (charEP 'A') 41 "BCD"

(42,Left "pos 42: unexpected B")

> runPrsEP (charEP 'A') 41 ""

(42,Left "pos 42: unexpected end of input")

Вспомогательная функция parseEP дает возможность вызывать парсер более удобным образом по сравнению с runPrsEP, скрывая технические детали:

GHCi> parseEP (charEP 'A') "ABC"

Right ('A',"BC")

GHCi> parseEP (charEP 'A') "BCD"

Left "pos 1: unexpected B"

GHCi> parseEP (charEP 'A') ""

Left "pos 1: unexpected end of input"

Реализуйте функцию satisfyEP :: (Char -> Bool) -> PrsEP Char, обеспечивающую описанное выше поведение.

Ответ:

{-newtype PrsEP a = PrsEP { runPrsEP :: Int -> String -> (Int, Either String (a, String)) }

parseEP :: PrsEP a -> String -> Either String (a, String)

parseEP p = snd . runPrsEP p 0-}

satisfyEP :: (Char -> Bool) -> PrsEP Char

satisfyEP p = PrsEP $ f . succ where

f pos (s:ls) | p s = (pos, Right (s, ls))

| otherwise = formatError pos $ ": unexpected " ++ [s]

f pos \_ = formatError pos ": unexpected end of input"

formatError pos str = (pos, Left $ "pos " ++ show pos ++ str)

--charEP c = satisfyEP (== c)

### Шаг 8:

Сделайте парсер

newtype PrsEP a = PrsEP { runPrsEP :: Int -> String -> (Int, Either String (a, String)) }

parseEP :: PrsEP a -> String -> Either String (a, String)

parseEP p = snd . runPrsEP p 0

представителем классов типов Functor и Applicative, обеспечив следующее поведение:

GHCi> runPrsEP (pure 42) 0 "ABCDEFG"

(0,Right (42,"ABCDEFG"))

GHCi> charEP c = satisfyEP (== c)

GHCi> anyEP = satisfyEP (const True)

GHCi> testP = (,) <$> anyEP <\* charEP 'B' <\*> anyEP

GHCi> runPrsEP testP 0 "ABCDE"

(3,Right (('A','C'),"DE"))

GHCi> parseEP testP "BCDE"

Left "pos 2: unexpected C"

GHCi> parseEP testP ""

Left "pos 1: unexpected end of input"

GHCi> parseEP testP "B"

Left "pos 2: unexpected end of input"

Ответ:

import Control.Applicative(liftA)

--newtype PrsEP a = PrsEP { runPrsEP :: Int -> String -> (Int, Either String (a, String)) }

instance Functor PrsEP where

fmap = liftA

instance Applicative PrsEP where

pure v = PrsEP (\str pos -> (str, Right (v, pos)))

(<\*>) f p = bind f $ bind p . (pure .) where {-ap for bind-}

bind (PrsEP p) pf = PrsEP $ (bind'. ) . p where

bind' (pos, Right (val, str)) = runPrsEP (pf val) pos str

bind' (pos, Left str) = (pos, Left str)

{-instance Functor PrsEP where

fmap = liftM

instance Applicative PrsEP where

pure = return

(<\*>) = ap

instance Monad PrsEP where

return x = PrsEP (\s pos->(s, Right (x, pos)))

(>>=) (PrsEP p) pf = PrsEP $ (f .) . p where

f (pos, Right (val, str)) = runPrsEP (pf val) pos str

f (pos, Left str) = (pos, Left str)-}

### Шаг 9:

Сделайте парсер

newtype PrsEP a = PrsEP { runPrsEP :: Int -> String -> (Int, Either String (a, String)) }

parseEP :: PrsEP a -> String -> Either String (a, String)

parseEP p = snd . runPrsEP p 0

представителем класса типов Alternative, обеспечив следующее поведение для пары неудачных альтернатив: сообщение об ошибке возвращается из той альтернативы, которой удалось распарсить входную строку глубже.

GHCi> runPrsEP empty 0 "ABCDEFG"

(0,Left "pos 0: empty alternative")

GHCi> charEP c = satisfyEP (== c)

GHCi> tripleP [a,b,c] = (\x y z -> [x,y,z]) <$> charEP a <\*> charEP b <\*> charEP c

GHCi> parseEP (tripleP "ABC" <|> tripleP "ADC") "ABE"

Left "pos 3: unexpected E"

GHCi> parseEP (tripleP "ABC" <|> tripleP "ADC") "ADE"

Left "pos 3: unexpected E"

GHCi> parseEP (tripleP "ABC" <|> tripleP "ADC") "AEF"

Left "pos 2: unexpected E"

Ответ:

import Control.Applicative(Alternative(..))

instance Alternative PrsEP where

empty = PrsEP (\pos \_-> (pos, Left $ "pos " ++ show pos ++ ": empty alternative"))

(<|>) (PrsEP l) (PrsEP r) = PrsEP (\pos str -> pats (l pos str) (r pos str)) where

pats l@(\_,Right (\_,\_)) r = l

pats l r@(\_,Right (\_,\_)) = r

pats l@(lp,\_) r@(rp,\_) = if lp>=rp then l else r

# МОНАДЫ И ЭФФЕКТЫ

## Монада Except

### Шаг 3:

Реализуйте функцию withExcept :: (e -> e') -> Except e a -> Except e' a, позволящую, если произошла ошибка, применить к ней заданное преобразование.

Ответ:

--newtype Except e a = Except {runExcept :: Either e a} deriving Show

--except = Except

withExcept :: (e -> e') -> Except e a -> Except e' a

withExcept f e = case runExcept e of

Right ok -> except $ Right ok

Left err -> except $ Left $ f err

### Шаг 7:

В модуле Control.Monad.Trans.Except библиотеки transformers имеется реализация монады Except с интерфейсом, идентичным представленному в видео-степах, но с более общими типами. Мы изучим эти типы в следующих модулях, однако использовать монаду Except из библиотеки transformers мы можем уже сейчас.

Введём тип данных для представления ошибки обращения к списку по недопустимому индексу:

data ListIndexError = ErrIndexTooLarge Int | ErrNegativeIndex

deriving (Eq, Show)

Реализуйте оператор (!!!) :: [a] -> Int -> Except ListIndexError a доступа к элементам массива по индексу, отличающийся от стандартного (!!) поведением в исключительных ситуациях. В этих ситуациях он должен выбрасывать подходящее исключение типа ListIndexError.

GHCi> runExcept $ [1..100] !!! 5

Right 6

GHCi> (!!!!) xs n = runExcept $ xs !!! n

GHCi> [1,2,3] !!!! 0

Right 1

GHCi> [1,2,3] !!!! 42

Left (ErrIndexTooLarge 42)

GHCi> [1,2,3] !!!! (-3)

Left ErrNegativeIndex

Ответ:

infixl 9 !!!

(!!!) :: [a] -> Int -> Except ListIndexError a

(!!!) s n | n<0 = throwE ErrNegativeIndex

| otherwise = except $ case drop n s of

h:\_ -> Right $ h

\_ -> Left $ ErrIndexToo

### Шаг 8:

Реализуйте функцию tryRead, получающую на вход строку и пытающуюся всю эту строку превратить в значение заданного типа. Функция должна возвращать ошибку в одном из двух случаев: если вход был пуст или если прочитать значение не удалось.

Информация об ошибке хранится в специальном типе данных:

data ReadError = EmptyInput | NoParse String

deriving Show

GHCi> runExcept (tryRead "5" :: Except ReadError Int)

Right 5

GHCi> runExcept (tryRead "5" :: Except ReadError Double)

Right 5.0

GHCi> runExcept (tryRead "5zzz" :: Except ReadError Int)

Left (NoParse "5zzz")

GHCi> runExcept (tryRead "(True, ())" :: Except ReadError (Bool, ()))

Right (True,())

GHCi> runExcept (tryRead "" :: Except ReadError (Bool, ()))

Left EmptyInput

GHCi> runExcept (tryRead "wrong" :: Except ReadError (Bool, ()))

Left (NoParse "wrong")

Ответ:

{-import Control.Monad.Trans.Except

data ReadError = EmptyInput | NoParse String deriving Show-}

tryRead :: Read a => String -> Except ReadError a

tryRead "" = throwE EmptyInput

tryRead s = case reads s of

[(v, "")] -> except $ Right v

\_ -> throwE $ NoParse s

### Шаг 9:

Используя tryRead из прошлого задания, реализуйте функцию trySum, которая получает список чисел, записанных в виде строк, и суммирует их. В случае неудачи, функция должна возвращать информацию об ошибке вместе с номером элемента списка (нумерация с единицы), вызвавшим ошибку.

Для хранения информации об ошибке и номере проблемного элемента используем новый тип данных:

data SumError = SumError Int ReadError

deriving Show

GHCi> runExcept $ trySum ["10", "20", "30"]

Right 60

GHCi> runExcept $ trySum ["10", "20", ""]

Left (SumError 3 EmptyInput)

GHCi> runExcept $ trySum ["10", "two", "30"]

Left (SumError 2 (NoParse "two"))

Подсказка: функция [withExcept](https://hackage.haskell.org/package/transformers-0.5.4.0/docs/Control-Monad-Trans-Except.html#v:withExcept) в этом задании может быть чрезвычайно полезна. Постарайтесь максимально эффективно применить знания, полученные на прошлой неделе.

Ответ:

{-import Control.Monad.Trans.Except

data ReadError = EmptyInput | NoParse String deriving Show

data SumError = SumError Integer ReadError deriving Show-}

trySum :: [String] -> Except SumError Integer

trySum = let merge (pos, item) e' = do

v<-withExcept (SumError pos) (tryRead item)

v'<-e'

return $ v+v'

in foldr merge (except $ Right 0) . zip [1..]

### Шаг 13:

Тип данных для представления ошибки обращения к списку по недопустимому индексу

data ListIndexError = ErrIndexTooLarge Int | ErrNegativeIndex

deriving (Eq, Show)

не очень естественно делать представителем класса типов Monoid. Однако, если мы хотим обеспечить накопление информации об ошибках, моноид необходим. К счастью, уже знакомая нам функция withExcept :: (e -> e') -> Except e a -> Except e' a позволяет изменять тип ошибки при вычислении в монаде Except.

Сделайте тип данных

newtype SimpleError = Simple { getSimple :: String }

deriving (Eq, Show)

представителем необходимых классов типов и реализуйте преобразователь для типа данных ошибки lie2se :: ListIndexError -> SimpleError так, чтобы обеспечить следующее поведение

GHCi> toSimple = runExcept . withExcept lie2se

GHCi> xs = [1,2,3]

GHCi> toSimple $ xs !!! 42

Left (Simple {getSimple = "[index (42) is too large]"})

GHCi> toSimple $ xs !!! (-2)

Left (Simple {getSimple = "[negative index]"})

GHCi> toSimple $ xs !!! 2

Right 3

GHCi> import Data.Foldable (msum)

GHCi> toSimpleFromList = runExcept . msum . map (withExcept lie2se)

GHCi> toSimpleFromList [xs !!! (-2), xs !!! 42]

Left (Simple {getSimple = "[negative index][index (42) is too large]"})

GHCi> toSimpleFromList [xs !!! (-2), xs !!! 2]

Right 3

Ответ:

import Data.Function (on)

{-data ListIndexError = ErrIndexTooLarge Int | ErrNegativeIndex deriving (Eq, Show)

newtype SimpleError = Simple { getSimple :: String } deriving (Eq, Show)-}

instance Monoid SimpleError where

mempty = Simple ""

mappend = on ((Simple .) . (++)) getSimple

lie2se :: ListIndexError -> SimpleError

lie2se (ErrIndexTooLarge i) = Simple $ "[index (" ++ show i ++ ") is too large]"

lie2se (ErrNegativeIndex) = Simple "[negative index]"

### Шаг 14:

Стандартная семантика Except как аппликативного функтора и монады: выполнять цепочку вычислений до первой ошибки. Реализация представителей классов Alternative и MonadPlus наделяет эту монаду альтернативной☺ семантикой: попробовать несколько вычислений, вернуть результат первого успешного, а в случае неудачи — все возникшие ошибки.

Довольно часто возникает необходимость сделать нечто среднее. К примеру, при проверке корректности заполнения анкеты или при компиляции программы для общего успеха необходимо, чтобы ошибок совсем не было, но в то же время, нам хотелось бы не останавливаться после первой же ошибки, а продолжить проверку, чтобы отобразить сразу все проблемы. Except такой семантикой не обладает, но никто не может помешать нам сделать свой тип данных (назовем его Validate), представители которого будут обеспечивать требую семантику, позволяющую сохранить список всех произошедших ошибок:

newtype Validate e a = Validate { getValidate :: Either [e] a }

Реализуйте функцию validateSum :: [String] -> Validate SumError Integer:

GHCi> getValidate $ validateSum ["10", "20", "30"]

Right 60

GHCi> getValidate $ validateSum ["10", "", "30", "oops"]

Left [SumError 2 EmptyInput,SumError 4 (NoParse "oops")]

Эта функция практически ничем не отличается от уже реализованной ранее trySum, если использовать функцию-адаптер collectE :: Except e a -> Validate e a и представителей каких-нибудь классов типов для Validate.

Ответ:

import Control.Monad.Trans.Except

import Control.Applicative(liftA)

--newtype Validate e a = Validate { getValidate :: Either [e] a } deriving Show

throwV = Validate . Left

instance Functor (Validate e) where

fmap = liftA

instance Applicative (Validate e) where

pure = Validate . Right

Validate (Right f) <\*> Validate (Right v) = Validate $ Right $ f v

Validate (Left s) <\*> Validate (Left v) = throwV $ s++v

Validate (Left f) <\*> \_ = throwV f

\_ <\*> Validate (Left v) = throwV v

collectE :: Except e a -> Validate e a

collectE v = Validate $ either (Left . pure) Right $ runExcept v

validateSum :: [String] -> Validate SumError Integer

validateSum ls = sum <$> traverse fn (zip [1..] ls) where

fn (pos, it) = collectE $ withExcept (SumError pos) $ tryRead it

## Монада Cont

### Шаг 3:

CPS-преобразование часто применяют для создания [предметно-ориентированных языков](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%B4%D0%BC%D0%B5%D1%82%D0%BD%D0%BE-%D0%BE%D1%80%D0%B8%D0%B5%D0%BD%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) ([DSL](https://en.wikipedia.org/wiki/Domain-specific_language)).

Реализуйте комбинаторы, которые позволят записывать числа вот в таком забавном формате:

GHCi> decode one hundred twenty three as a number

123

GHCi> decode one hundred twenty one as a number

121

GHCi> decode one hundred twenty as a number

120

GHCi> decode one hundred as a number

100

GHCi> decode three hundred as a number

300

GHCi> decode two thousand seventeen as a number

2017

*Достаточно чтобы работали всякие простые случаи, как в примерах; не старайтесь поддержать прямо все допустимые варианты, это потребует несколько бóльших усилий.*

Ответ:

import Data.Function((&))

decode = (&) 0

as = const . const

a = id

number = id

one = (&) . (+1)

two = (&) . (+2)

three = (&) . (+3)

seventeen = (&) . (+17)

twenty = (&) . (+20)

hundred = (&) . (\*100)

thousand = (&) . (\*1000)

{- 1) one v f = f $ (1+) v

2) one v f = (1+) v & f

3) one v f =(&) ((1+) v) f -- target step, now f is ends expression

4) one v =(&) ((1+) v)

5) one =(&) . (1+)-}

### Шаг 6:

Реализуйте функцию showCont, запускающую вычисление и возвращающую его результат в виде строки.

Ответ:

showCont :: Show a => Cont String a -> String

showCont = flip runCont show

### Шаг 9:

Возможность явно работать с продолжением обеспечивает доступ к очень гибкому управлению исполнением. В этом задании вам предстоит реализовать вычисление, которое анализирует и модифицирует значение, возвращаемое кодом, написанным *после* него.

В качестве примера рассмотрим следующую функцию:

addTens :: Int -> Checkpointed Int

addTens x1 = \checkpoint -> do

checkpoint x1

let x2 = x1 + 10

checkpoint x2 {- x2 = x1 + 10 -}

let x3 = x2 + 10

checkpoint x3 {- x3 = x1 + 20 -}

let x4 = x3 + 10

return x4 {- x4 = x1 + 30 -}

Эта функция принимает значение x1, совершает с ним последовательность операций (несколько раз прибавляет 10) и после каждой операции «сохраняет» промежуточный результат. При запуске такой функции используется дополнительный предикат, который является критерием «корректности» результата, и в случае, если возвращенное функцией значение этому критерию не удовлетворяет, вернется последнее удовлетворяющее ему значение из «сохраненных»:

GHCi> runCheckpointed (< 100) $ addTens 1

31

GHCi> runCheckpointed (< 30) $ addTens 1

21

GHCi> runCheckpointed (< 20) $ addTens 1

11

GHCi> runCheckpointed (< 10) $ addTens 1

1

(Если ни возвращенное, ни сохраненные значения не подходят, результатом должно быть первое из сохраненных значений; если не было сохранено ни одного значения, то результатом должно быть возвращенное значение.)

Обратите внимание на то, что функция checkpoint передается в Checkpointed вычисление как параметр, поскольку её поведение зависит от предиката, который будет известен только непосредственно при запуске.

Ответ:

--import Control.Monad.Trans.Cont

type Checkpointed a = (a -> Cont a a) -> (Cont a a)

runCheckpointed :: (a -> Bool) -> Checkpointed a -> a

runCheckpointed pd ck = runCont (ck helper) id where

helper cur = {-cont-}Cont (\f -> let next = f cur in

if pd next then next else cur)

{-addTens :: Int -> Checkpointed Int

addTens x1 = \checkpoint ->

let

x2 = x1 + 10

x3 = x2 + 10

x4 = x3 + 10

in checkpoint x1 >> checkpoint x2 >> checkpoint x3 >> return x4

main = print $ runCheckpointed (\x -> x<21) $ addTens 4

-}

{-

addTens :: Int -> Checkpointed Int

addTens x1 = \checkpoint -> do

checkpoint x1

let x2 = x1 + 10

checkpoint x2

let x3 = x2 + 10

checkpoint x3

let x4 = x3 + 10

return x4

Рассуждения по нахождению сигнатуры для типа type Checkpointed a = ?

Упростим do-нотацию в пользу операторов монадического связывания

addTens :: Int -> Checkpointed Int

addTens x1 = \checkpoint ->

let

x2 = x1 + 10

x3 = x2 + 10

x4 = x3 + 10

in checkpoint x1 >> checkpoint x2 >> checkpoint x3 >> return x4

Заметим нижеперичисленное:

1. Так как определение Функции addTens начинается с лямбды, то возвращаемый тип - функция.

? -> ? (Далее целевая сигнатура)

2. С точки зрения целевой функции, её первый аргумент(в этой области) назван checkpoint.

Далее по коду, checkpoint фигурирует в цепи:

checkpoint x1 >> checkpoint x2 >> checkpoint x3 >> return x4

Фактические, выполняется вызов checkpoint с целочисленным числом,

а каждое полученное выражения связано оператором >>.

Внесём извлеченную информацию в целевую сигнатуру:

(Int -> SomeMonad) -> SomeMonad

3. Воспользуемся предположением, согласно которому, SomeMonad является монадой Cont:

(Int -> Cont ? ?) -> Cont ? ?

4. TODO

-}

### Шаг 10:

Вычисление в монаде Cont передает результат своей работы в функцию-продолжение. А что, если наши вычисления могут завершиться с ошибкой? В этом случае мы могли бы явно возвращать значение типа Either и каждый раз обрабатывать два возможных исхода, что не слишком удобно. Более разумный способ решения этой проблемы предоставляют трансформеры монад, но с ними мы познакомимся немного позже.

Определите тип данных FailCont для вычислений, которые получают два продолжения и вызывают одно из них в случае успеха, а другое — при неудаче. Сделайте его представителем класса типов Monad и реализуйте вспомогательные функции toFailCont и evalFailCont, используемые в следующем коде:

add :: Int -> Int -> FailCont r e Int

add x y = FailCont $ \ok \_ -> ok $ x + y

addInts :: String -> String -> FailCont r ReadError Int

addInts s1 s2 = do

i1 <- toFailCont $ tryRead s1

i2 <- toFailCont $ tryRead s2

return $ i1 + i2

(Здесь используется функция tryRead из предыдущего урока; определять её заново не надо.)

GHCi> evalFailCont $ addInts "15" "12"

Right 27

GHCi> runFailCont (addInts "15" "") print (putStrLn . ("Oops: " ++) . show)

Oops: EmptyInput

Ответ:

import Control.Monad(liftM,ap)

import Control.Monad.Trans.Except

{-

Шаги по разрешению сигнатуры: newtype FailCont r e a = FailCont { runFailCont :: ???}

1) По условию дано: add x y = FailCont $ \ok \_ -> ok $ x + y

Что приводит к заключению о том, что искомая сигнатура - функция двух аргументов:

() -> () -> ()

2) Кроме того, вызов вида \ok \_ -> ok $ x + y, предоставляет достаточно информации

относительно первого параметра - он является функцией одного аргумента, при этом

его возвращаемый тип служит конечным типом всего выражения. Тогда:

(l->q) -> () -> q

3) В Условие задачи явно сказано, что искомая сигнатура должна обеспечивать два продолжения,

значит вторым аргументом также выступает функция:

(l->q) -> (->) -> q

4) Возвращаемый тип второй функции-аргумента должен совпадать с типом всего выражения, иначе

возможность строить продолжения окажется невозможной. Учтем это:

(l->q) -> (->q) -> q

5) Конструктор типа FailCont трехпараметравый, в текущий сигнатуре использовано только два,

в следствии входной тип второй функции-аргумента не совпадает ни с каким другим.

(l -> q) -> (r -> q) -> q

newtype FailCont r e a = FailCont { runFailCont :: (a -> r) -> (e -> r) -> r}

-}

newtype FailCont r e a = FailCont { runFailCont :: (a -> r) -> (e -> r) -> r}

instance Functor (FailCont r e) where

fmap = liftM

instance Applicative (FailCont r e) where

pure = return

(<\*>) = ap

instance Monad (FailCont r e) where

{-return:: a -> c a

a -> FailCont r e a-}

return v = FailCont $ (\ok \_ -> ok v)

{-(>>=):: c a -> (a -> c b) -> c b

FailCont r e a -> (a -> FailCont r e b) -> (FailCont r e b)

((a -> r) -> (e -> r) -> r) -> (a -> ((b -> r) -> (e -> r) -> r)) -> ((b -> r) -> (e -> r) -> r)-}

(>>=) (FailCont v) c = FailCont (\ok er -> v (\a->runFailCont (c a) ok er) er)

toFailCont :: Except e a -> FailCont r e a

toFailCont = either (\l-> FailCont $ (\\_ er -> er l)) return . runExcept

evalFailCont :: FailCont (Either e a) e a -> Either e a

evalFailCont (FailCont v) = v Right Left

### Шаг 13:

Реализуйте функцию callCFC для монады FailCont по аналогии с callCC.

Ответ:

callCFC :: ((a -> FailCont r e b) -> FailCont r e a) -> FailCont r e a

{- (a -> r) -> (e -> r) -> r

((a -> FailCont r e b) -> FailCont r e a) -> FailCont r e a

((a -> ((b -> r) -> (e -> r) -> r)) -> ((a -> r) -> (e -> r) -> r)) -> ((a -> r) -> (e -> r) -> r) -}

callCFC f = FailCont (\a'r -> runFailCont (f (\a -> FailCont (\\_ \_ -> a'r a))) a'r)

{- Кавычка в названии параметра отождествляет функцию

callCFC f =

1) FailCont (\a'r e'r-> runFailCont (f (undefined)) a'r e'r)

2) FailCont (\a'r e'r-> runFailCont (f (\a'brerr-> undefined)) a'r e'r)

3) FailCont (\a'r e'r-> runFailCont (f (\a'brerr-> FailCont (\b'r e'r->undefined))) a'r e'r)

4) FailCont (\a'r e'r-> runFailCont (f (\a{-brerr-} -> FailCont (\b'r e'r -> a'r a))) a'r e'r)

5) FailCont (\a'r e'r-> runFailCont (f (\a -> FailCont (\\_ \_ -> a'r a))) a'r e'r)-}

## Трансформеры монад

### Шаг 4:

Перепишите функцию logFirstAndRetSecond из предыдущего видео, используя трансформер WriterT из модуля Control.Monad.Trans.Writer библиотеки transformers, и монаду Reader в качестве базовой.

GHCi> runReader (runWriterT logFirstAndRetSecond) strings

("DEFG","abc")

Ответ:

{-for ghc 8.0.2

import Control.Monad.Trans.Class(lift)

import Control.Monad.Trans.Reader(Reader, asks, runReader)

import Control.Monad.Trans.Writer(WriterT, tell, runWriterT)

import Data.Char (toUpper)-}

import Control.Monad.Trans.Reader(Reader, asks, runReader)

import Control.Monad.Trans.Writer(WriterT, tell, runWriterT)

import Control.Monad.Trans(MonadTrans(lift))

import Data.Char(toUpper)

logFirstAndRetSecond :: WriterT String (Reader [String]) String

logFirstAndRetSecond = do

h <- lift $ asks head

tell h

lift $ asks (map toUpper . head . tail)

### Шаг 5:

Реализуйте функцию separate :: (a -> Bool) -> (a -> Bool) -> [a] -> WriterT [a] (Writer [a]) [a].

Эта функция принимает два предиката и список и записывает в один лог элементы списка, удовлетворяющие первому предикату, в другой лог — второму предикату, а возвращающает список элементов, ни одному из них не удовлетворяющих.

GHCi> (runWriter . runWriterT) $ separate (<3) (>7) [0..10]

(([3,4,5,6,7],[0,1,2]),[8,9,10])

Ответ:

import Control.Monad(when)

separate :: (a -> Bool) -> (a -> Bool) -> [a] -> WriterT [a] (Writer [a]) [a]

separate \_ \_ [] = return []

separate p1 p2 (s:ls) = when (p1 s) (tell [s]) >>

when (p2 s) (lift $ tell [s]) >>

separate p1 p2 ls >>= (\acc->

return $ if (p1 s) || (p2 s) then acc else s:acc)

### Шаг 7:

Наша абстракция пока что недостаточно хороша, поскольку пользователь всё ещё должен помнить такие детали, как, например, то, что asks нужно вызывать напрямую, а tell — только с помощью lift.

Нам хотелось бы скрыть такие детали реализации, обеспечив унифицированный интерфейс доступа к возможностям нашей монады, связанным с чтением окружения, и к возможностям, связанным с записью в лог. Для этого реализуйте функции myAsks и myTell, позволяющие записать logFirstAndRetSecond следующим образом:

logFirstAndRetSecond :: MyRW String

logFirstAndRetSecond = do

el1 <- myAsks head

el2 <- myAsks (map toUpper . head . tail)

myTell el1

return el2

Ответ:

myAsks :: ([String] -> a) -> MyRW a

myAsks = asks

myTell :: String -> MyRW ()

myTell = lift . tell

### Шаг 9:

Превратите монаду MyRW в трансформер монад MyRWT:

logFirstAndRetSecond :: MyRWT IO String

logFirstAndRetSecond = do

el1 <- myAsks head

myLift $ putStrLn $ "First is " ++ show el1

el2 <- myAsks (map toUpper . head . tail)

myLift $ putStrLn $ "Second is " ++ show el2

myTell el1

return el2

GHCi> runMyRWT logFirstAndRetSecond ["abc","defg","hij"]

First is "abc"

Second is "DEFG"

("DEFG","abc")

Ответ:

{---for ghc 8.0.2

import Control.Monad.Trans.Class(lift)

import Control.Monad.Trans.Reader

import Control.Monad.Trans.Writer

import Data.Char (toUpper)-}

type MyRWT m = ReaderT [String] (WriterT String m)

runMyRWT :: MyRWT m t -> [String] -> m (t,String)

runMyRWT = (runWriterT .) . runReaderT

--runMyRWT rw ls = runWriterT (runReaderT rw ls)

myAsks :: Monad m => ([String] -> a) -> MyRWT m a

myAsks = asks

myTell :: Monad m => (String -> MyRWT m ())

myTell = lift . tell

myLift :: Monad m => m a -> MyRWT m a

myLift = lift . lift

### Шаг 10:

С помощью трансформера монад MyRWT мы можем написать *безопасную* версию logFirstAndRetSecond:

logFirstAndRetSecond :: MyRWT Maybe String

logFirstAndRetSecond = do

xs <- myAsk

case xs of

(el1 : el2 : \_) -> myTell el1 >> return (map toUpper el2)

\_ -> myLift Nothing

GHCi> runMyRWT logFirstAndRetSecond ["abc","defg","hij"]

Just ("DEFG","abc")

GHCi> runMyRWT logFirstAndRetSecond ["abc"]

Nothing

Реализуйте *безопасную* функцию veryComplexComputation, записывающую в лог через запятую первую строку четной длины и первую строку нечетной длины, а возвращающую пару из второй строки четной и второй строки нечетной длины, приведенных к верхнему регистру:

GHCi> runMyRWT veryComplexComputation ["abc","defg","hij"]

Nothing

GHCi> runMyRWT veryComplexComputation ["abc","defg","hij","kl"]

Just (("KL","HIJ"),"defg,abc")

Подсказка: возможно, полезно будет реализовать функцию [myWithReader](https://hackage.haskell.org/package/transformers-0.5.4.0/docs/Control-Monad-Trans-Reader.html#v:withReader).

Ответ:

{-

import Control.Monad.Trans.Class(lift)

import Control.Monad.Trans.Reader

import Control.Monad.Trans.Writer

import Data.Char (toUpper)

type MyRWT m = ReaderT [String] (WriterT String m)

runMyRWT :: MyRWT m t -> [String] -> m (t,String)

runMyRWT = (runWriterT .) . runReaderT

myAsks :: Monad m => ([String] -> a) -> MyRWT m a

myAsks = asks

myAsk = ask

myTell :: Monad m => (String -> MyRWT m ())

myTell = lift . tell

myLift :: Monad m => m a -> MyRWT m a

myLift = lift . lift

-}

veryComplexComputation :: MyRWT Maybe (String, String)

veryComplexComputation = local (\s->"":"":s) compute where

recall x = local(\\_->x) compute

check s s' = even (length s) /= even (length s')

swap s s' = if even (length s) then (s,s') else (s',s)

wr s s' = let (p,p')=swap s s' in p++',':p'

compute = do

lg:rt:ls<-myAsk

case ls of

(s:ls) -> if null lg then recall (('!':s):rt:ls)

else if head lg=='!'&&check s (tail lg)then myTell(wr s $ tail lg)>>recall (".":rt:ls)

else if null rt then recall (lg:(' ':s):ls)

else if check s (tail rt) then return (swap (map toUpper s) (map toUpper $ tail rt))

else recall (lg:rt:ls)

[] -> myLift Nothing

### Шаг 11:

Предположим мы хотим исследовать свойства рекуррентных последовательностей. Рекуррентные отношения будем задавать вычислениями типа State Integer Integer, которые, будучи инициализированы текущим значением элемента последовательности, возвращают следующее значение в качестве состояния и текущее в качестве возвращаемого значения, например:

tickCollatz :: State Integer Integer

tickCollatz = do

n <- get

let res = if odd n then 3 \* n + 1 else n `div` 2

put res

return n

Используя монаду State из модуля Control.Monad.Trans.State и трансформер ExceptT из модуля Control.Monad.Trans.Except библиотеки transformers, реализуйте для монады

type EsSi = ExceptT String (State Integer)

функцию runEsSi :: EsSi a -> Integer -> (Either String a, Integer), запускающую вычисление в этой монаде, а также функцию go :: Integer -> Integer -> State Integer Integer -> EsSi (), принимающую шаг рекуррентного вычисления и два целых параметра, задающие нижнюю и верхнюю границы допустимых значений вычислений. Если значение больше или равно верхнему или меньше или равно нижнему, то оно прерывается исключением с соответствующим сообщением об ошибке

GHCi> runEsSi (go 1 85 tickCollatz) 27

(Right (),82)

GHCi> runEsSi (go 1 80 tickCollatz) 27

(Left "Upper bound",82)

GHCi> runEsSi (forever $ go 1 1000 tickCollatz) 27

(Left "Upper bound",1186)

GHCi> runEsSi (forever $ go 1 10000 tickCollatz) 27

(Left "Lower bound",1)

Ответ:

import Control.Monad

import Control.Monad.Trans

--For ghci 8.0.2 import Control.Monad.Trans.Class

import Control.Monad.Trans.State

import Control.Monad.Trans.Except

tickCollatz :: State Integer Integer

tickCollatz = do

n <- get

let res = if odd n then 3 \* n + 1 else n `div` 2

put res

return n

type EsSi = ExceptT String (State Integer)

runEsSi :: EsSi a -> Integer -> (Either String a, Integer)

runEsSi = runState . runExceptT

go :: Integer -> Integer -> State Integer Integer -> EsSi ()

go l u st = do

s <- lift $ get

let next = execState st s

lift $ put next

when (next<=l) $ throwE "Lower bound"

when (next>=u) $ throwE "Upper bound"

### Шаг 12:

Модифицируйте монаду EsSi из предыдущей задачи, обернув ее в трансформер ReaderT с окружением, представляющим собой пару целых чисел, задающих нижнюю и верхнюю границы для вычислений. Функции go теперь не надо будет передавать эти параметры, они будут браться из окружения. Сделайте получившуюся составную монаду трансформером:

type RiiEsSiT m = ReaderT (Integer,Integer) (ExceptT String (StateT Integer m))

Реализуйте также функцию для запуска этого трансформера

runRiiEsSiT :: ReaderT (Integer,Integer) (ExceptT String (StateT Integer m)) a

-> (Integer,Integer)

-> Integer

-> m (Either String a, Integer)

и модифицируйте код функции go, изменив её тип на

go :: Monad m => StateT Integer m Integer -> RiiEsSiT m ()

так, чтобы для шага вычисления последовательности с отладочным выводом текущего элемента последовательности на экран

tickCollatz' :: StateT Integer IO Integer

tickCollatz' = do

n <- get

let res = if odd n then 3 \* n + 1 else n `div` 2

lift $ putStrLn $ show res

put res

return n

мы получили бы

GHCi> runRiiEsSiT (forever $ go tickCollatz') (1,200) 27

82

41

124

62

31

94

47

142

71

214

(Left "Upper bound",214)

Ответ:

import Control.Monad

import Control.Monad.Trans

import Control.Monad.Trans.Reader

import Control.Monad.Trans.State

import Control.Monad.Trans.Except

type RiiEsSiT m = ReaderT (Integer,Integer) (ExceptT String (StateT Integer m))

runRiiEsSiT :: ReaderT (Integer,Integer) (ExceptT String (StateT Integer m)) a

-> (Integer,Integer)

-> Integer

-> m (Either String a, Integer)

runRiiEsSiT m lu n = runStateT (runExceptT $ runReaderT m lu) n

go :: Monad m => StateT Integer m Integer -> RiiEsSiT m ()

go st = do

lift $ lift st

next <- lift $ lift get

~(l,u) <- ask

when (next<=l) $ lift $ throwE "Lower bound"

when (next>=u) $ lift $ throwE "Upper bound"

## Трансформер ReaderT

### Шаг 3:

В задачах из предыдущих модулей мы сталкивались с типами данных

newtype Arr2 e1 e2 a = Arr2 { getArr2 :: e1 -> e2 -> a }

newtype Arr3 e1 e2 e3 a = Arr3 { getArr3 :: e1 -> e2 -> e3 -> a }

задающих вычисления с двумя и тремя окружениями соответственно. Можно расширить их до трансформеров:

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

Напишите «конструирующие» функции

arr2 :: Monad m => (e1 -> e2 -> a) -> Arr2T e1 e2 m a

arr3 :: Monad m => (e1 -> e2 -> e3 -> a) -> Arr3T e1 e2 e3 m a

обеспечивающие следующее поведение

GHCi> (getArr2T $ arr2 (+)) 33 9 :: [Integer]

[42]

GHCi> (getArr3T $ arr3 foldr) (\*) 1 [1..5] :: Either String Integer

Right 120

GHCi> import Data.Functor.Identity

GHCi> runIdentity $ (getArr2T $ arr2 (+)) 33 9

42

Ответ:

--newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

--newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

arr2 :: Monad m => (e1 -> e2 -> a) -> Arr2T e1 e2 m a

arr2 f = Arr2T $ (return .) . f

{- Arr2T (\x y-> return $ f x y)

Arr2T (\x-> return . (f x))-}

arr3 :: Monad m => (e1 -> e2 -> e3 -> a) -> Arr3T e1 e2 e3 m a

arr3 f = Arr3T $ ((return .) .) . f

{- Arr3T (\x y z -> return $ f x y z)

Arr3T (\x y -> return . (f x y))-}

### Шаг 6:

Сделайте трансформеры

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

представителями класса типов Functor в предположении, что m является функтором:

GHCi> a2l = Arr2T $ \e1 e2 -> [e1,e2,e1+e2]

GHCi> (getArr2T $ succ <$> a2l) 10 100

[11,101,111]

GHCi> a3e = Arr3T $ \e1 e2 e3 -> Right (e1+e2+e3)

GHCi> (getArr3T $ sqrt <$> a3e) 2 3 4

Right 3.0

Ответ:

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

instance Functor m => Functor (Arr2T e1 e2 m) where

{-

fmap:: (a->b) -> c a -> c b

(a->b) -> Arr2T e1 e2 m a -> Arr2T e1 e2 m b

(a->b) -> (e1 -> e2 -> m a) -> (e1 -> e2 -> m b)

-}

fmap f v = Arr2T $ (fmap f .) . getArr2T v

--fmap f (Arr2T v) = Arr2T (\e1 e2 ->fmap f (v e1 e2))

instance Functor m => Functor (Arr3T e1 e2 e3 m) where

fmap f v = Arr3T $ ((fmap f .) .) . getArr3T v

--fmap f (Arr3T v) = Arr3T (\e1 e2 e3 ->fmap f (v e1 e2 e3))

### Шаг 10:

Сделайте трансформеры

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

представителями класса типов Applicative в предположении, что m является аппликативным функтором:

GHCi> a2l = Arr2T $ \e1 e2 -> [e1,e2]

GHCi> a2fl = Arr2T $ \e1 e2 -> [(e1\*e2+),const 7]

GHCi> getArr2T (a2fl <\*> a2l) 2 10

[22,30,7,7]

GHCi> a3fl = Arr3T $ \e1 e2 e3 -> [(e2+),(e3+)]

GHCi> a3l = Arr3T $ \e1 e2 e3 -> [e1,e2]

GHCi> getArr3T (a3fl <\*> a3l) 3 5 7

[8,10,10,12]

Ответ:

import Control.Applicative(liftA2)

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

instance Functor m => Functor (Arr2T e1 e2 m) where

fmap f v = Arr2T $ (fmap f .) . getArr2T v

instance Functor m => Functor (Arr3T e1 e2 e3 m) where

fmap f v = Arr3T $ ((fmap f .) .) . getArr3T v

instance Applicative m => Applicative (Arr2T e1 e2 m) where

{-

(<\*>):: c (a->b) -> c a -> c b

Arr3T e1 e2 e3 m (a->b) -> Arr3T e1 e2 e3 m a -> Arr3T e1 e2 e3 m b

(e1 -> e2 -> m (a->b)) -> (e1 -> e2 -> m a) -> (e1 -> e2 -> m b)

-}

pure v = Arr2T (\\_ \_ -> pure v)

--pure = Arr2T . ((const . const) . pure)

--(<\*>) (Arr2T f) (Arr2T v) = Arr2T (\e1 e2 -> (f e1 e2)<\*>(v e1 e2))

(<\*>) f v = Arr2T $ (liftA2 . liftA2) (<\*>) (getArr2T f) (getArr2T v)

instance Applicative m => Applicative (Arr3T e1 e2 e3 m) where

pure v = Arr3T (\\_ \_ \_ -> pure v)

--pure = Arr3T . ((const . const . const) . pure)

--(<\*>) (Arr3T f) (Arr3T v) = Arr3T $ (\e1 e2 e3-> (f e1 e2 e3)<\*>(v e1 e2 e3))

(<\*>) f v = Arr3T $ (liftA2 . liftA2 . liftA2) (<\*>) (getArr3T f) (getArr3T v)

### Шаг 12:

Сделайте трансформеры

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

представителями класса типов Monad в предположении, что m является монадой:

GHCi> a2l = Arr2T $ \e1 e2 -> [e1,e2]

GHCi> getArr2T (do {x <- a2l; y <- a2l; return (x + y)}) 3 5

[6,8,8,10]

GHCi> a3m = Arr3T $ \e1 e2 e3 -> Just (e1 + e2 + e3)

GHCi> getArr3T (do {x <- a3m; y <- a3m; return (x \* y)}) 2 3 4

Just 81

Ответ:

import Control.Applicative(liftA2)

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

instance Functor m => Functor (Arr2T e1 e2 m) where

fmap f v = Arr2T $ (fmap f .) . getArr2T v

instance Functor m => Functor (Arr3T e1 e2 e3 m) where

fmap f v = Arr3T $ ((fmap f .) .) . getArr3T v

instance Applicative m => Applicative (Arr2T e1 e2 m) where

pure x = Arr2T (\\_ \_ -> pure x)

(<\*>) f v = Arr2T $ (liftA2 . liftA2) (<\*>) (getArr2T f) (getArr2T v)

instance Applicative m => Applicative (Arr3T e1 e2 e3 m) where

pure x = Arr3T (\\_ \_ \_ -> pure x)

(<\*>) f v = Arr3T $ (liftA2 . liftA2 . liftA2) (<\*>) (getArr3T f) (getArr3T v)

instance Monad m => Monad (Arr2T e1 e2 m) where

{-

(>>=) :: c a -> (a -> c b) -> c b

(>>=) :: (e1 -> e2 -> m a) -> (a -> (e1 -> e2 -> m b)) -> (e1 -> e2 -> m b)

-}

(>>=) v f = Arr2T (\e1 e2 -> getArr2T v e1 e2 >>= (\t-> getArr2T (f t) e1 e2))

fail v = Arr2T (\\_ \_ -> fail v)

instance Monad m => Monad (Arr3T e1 e2 e3 m) where

(>>=) v f = Arr3T (\e1 e2 e3 -> getArr3T v e1 e2 e3 >>= (\t-> getArr3T (f t) e1 e2 e3))

fail v = Arr3T (\\_ \_ \_ -> fail v)

### Шаг 13:

Разработанная нами реализация интерфейса монады для трансформера Arr3T (как и для Arr2T и ReaderT) имеет не очень хорошую особенность. При неудачном сопоставлении с образцом вычисления в этой монаде завершаются аварийно, с выводом сообщения об ошибке в диагностический поток:

GHCi> a3m = Arr3T $ \e1 e2 e3 -> Just (e1 + e2 + e3)

GHCi> getArr3T (do {9 <- a3m; y <- a3m; return y}) 2 3 4

Just 9

GHCi> getArr3T (do {10 <- a3m; y <- a3m; return y}) 2 3 4

\*\*\* Exception: Pattern match failure in do expression at :12:15-16

Для обычного ридера такое поведение нормально, однако у трансформера внутренняя монада может уметь обрабатывать ошибки более щадащим образом. Переопределите функцию fail класса типов Monad для Arr3T так, чтобы обработка неудачного сопоставления с образцом осуществлялась бы во внутренней монаде:

GHCi> getArr3T (do {10 <- a3m; y <- a3m; return y}) 2 3 4

Nothing

Ответ:

import Control.Applicative(liftA2)

newtype Arr3T e1 e2 e3 m a = Arr3T { getArr3T :: e1 -> e2 -> e3 -> m a }

instance Functor m => Functor (Arr3T e1 e2 e3 m) where

fmap f v = Arr3T $ ((fmap f .) .) . getArr3T v

instance Applicative m => Applicative (Arr3T e1 e2 e3 m) where

pure x = Arr3T (\\_ \_ \_ -> pure x)

(<\*>) f v = Arr3T $ (liftA2 . liftA2 . liftA2) (<\*>) (getArr3T f) (getArr3T v)

instance Monad m => Monad (Arr3T e1 e2 e3 m) where

(>>=) v f = Arr3T (\e1 e2 e3 -> getArr3T v e1 e2 e3 >>= (\t-> getArr3T (f t) e1 e2 e3))

fail v = Arr3T (\\_ \_ \_ -> fail v)

### Шаг 16:

Сделайте трансформер

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

представителями класса типов MonadTrans:

GHCi> a2l = Arr2T $ \e1 e2 -> [e1,e2]

GHCi> getArr2T (do {x <- a2l; y <- lift [10,20,30]; return (x+y)}) 3 4

[13,23,33,14,24,34]

Реализуйте также «стандартный интерфейс» для этой монады — функцию

asks2 :: Monad m => (e1 -> e2 -> a) -> Arr2T e1 e2 m a

работающую как asks для ReaderT, но принимающую при этом функцию от обоих наличных окружений:

GHCi> getArr2T (do {x <- asks2 const; y <- asks2 (flip const); z <- asks2 (,); return (x,y,z)}) 'A' 'B'

('A','B',('A','B'))

Ответ:

import Control.Applicative(liftA2)

--import Control.Monad.Trans.Class

class MonadTrans t where

lift :: Monad m => m a -> t m a

newtype Arr2T e1 e2 m a = Arr2T { getArr2T :: e1 -> e2 -> m a }

instance Functor m => Functor (Arr2T e1 e2 m) where

fmap f v = Arr2T $ (fmap f .) . getArr2T v

instance Applicative m => Applicative (Arr2T e1 e2 m) where

pure x = Arr2T (\\_ \_ -> pure x)

(<\*>) f v = Arr2T $ (liftA2 . liftA2) (<\*>) (getArr2T f) (getArr2T v)

instance Monad m => Monad (Arr2T e1 e2 m) where

(>>=) v f = Arr2T (\e1 e2 -> getArr2T v e1 e2 >>= (\t-> getArr2T (f t) e1 e2))

fail v = Arr2T (\\_ \_ -> fail v)

instance MonadTrans (Arr2T e1 e2) where

{-lift :: Monad m => m a -> t m a

m a -> Arr2T e1 e2 m a

m a -> (e1 -> e2 -> m a)-}

lift m = Arr2T(\\_ \_ -> m)

asks2 :: Monad m => (e1 -> e2 -> a) -> Arr2T e1 e2 m a

asks2 f = Arr2T $ (return .) . f

# ТРАНСФОРМЕРЫ МОНАД

## Трансформер WriterT

### Шаг 5:

Предположим, что мы дополнительно реализовали строгую версию аппликативного функтора Writer:

newtype StrictWriter w a = StrictWriter { runStrictWriter :: (a, w) }

instance Functor (StrictWriter w) where

fmap f = StrictWriter . updater . runStrictWriter

where updater (x, log) = (f x, log)

instance Monoid w => Applicative (StrictWriter w) where

pure x = StrictWriter (x, mempty)

f <\*> v = StrictWriter $ updater (runStrictWriter f) (runStrictWriter v)

where updater (g, w) (x, w') = (g x, w `mappend` w')

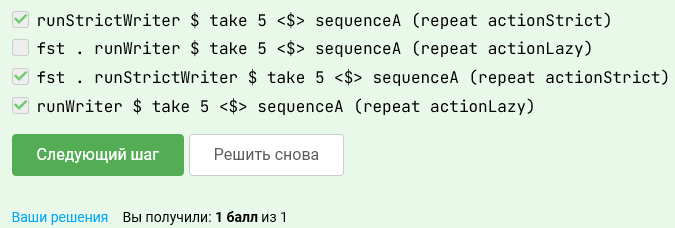
и определили такие вычисления:

actionLazy = Writer (42,"Hello!")

actionStrict = StrictWriter (42,"Hello!")

Какие из следующих вызовов приведут к расходимостям?

Ответ:



### Шаг 7:

Сделайте на основе типа данных

data Logged a = Logged String a deriving (Eq,Show)

трансформер монад LoggT :: (\* -> \*) -> \* -> \* с одноименным конструктором данных и меткой поля runLoggT:

newtype LoggT m a = LoggT { runLoggT :: m (Logged a) }

Для этого реализуйте для произвольной монады m представителя класса типов Monad для LoggT m :: \* -> \*:

instance Monad m => Monad (LoggT m) where

return x = undefined

m >>= k = undefined

fail msg = undefined

Для проверки используйте функции:

logTst :: LoggT Identity Integer

logTst = do

x <- LoggT $ Identity $ Logged "AAA" 30

y <- return 10

z <- LoggT $ Identity $ Logged "BBB" 2

return $ x + y + z

failTst :: [Integer] -> LoggT [] Integer

failTst xs = do

5 <- LoggT $ fmap (Logged "") xs

LoggT [Logged "A" ()]

return 42

которые при правильной реализации монады должны вести себя так:

GHCi> runIdentity (runLoggT logTst)

Logged "AAABBB" 42

GHCi> runLoggT $ failTst [5,5]

[Logged "A" 42,Logged "A" 42]

GHCi> runLoggT $ failTst [5,6]

[Logged "A" 42]

GHCi> runLoggT $ failTst [7,6]

[]

Ответ:

import Data.Monoid((<>))

import Control.Applicative(liftA2)

instance Functor m => Functor (LoggT m) where

fmap f v = LoggT $ fmap updater (runLoggT v)

where updater ~(Logged s v') = Logged s $ f v'

instance Applicative m => Applicative (LoggT m) where

pure v = LoggT $ pure $ Logged mempty v

f <\*> v = LoggT $ liftA2 updater (runLoggT f) (runLoggT v)

where updater ~(Logged s f) ~(Logged s' a) = Logged (s <> s') (f a)

instance Monad m => Monad (LoggT m) where

{-

>>= :: c a -> (a -> c b) -> c b

>>= :: LoggT m a -> (a -> LoggT m b) -> LoggT m b

m (Logged a) -> (a -> m (Logged a)) -> (m (Logged a))

-}

v >>= f = LoggT $ do

~(Logged s a) <- runLoggT v

~(Logged s' a') <- runLoggT $ f a

return $ Logged (s<>s') a'

fail = LoggT . fail

### Шаг 10:

Напишите функцию write2log обеспечивающую трансформер LoggT стандартным логгирующим интерфейсом:

write2log :: Monad m => String -> LoggT m ()

write2log = undefined

Эта функция позволяет пользователю осуществлять запись в лог в процессе вычисления в монаде LoggT m для любой монады m. Введите для удобства упаковку для LoggT Identity и напишите функцию запускающую вычисления в этой монаде

type Logg = LoggT Identity

runLogg :: Logg a -> Logged a

runLogg = undefined

Тест

logTst' :: Logg Integer

logTst' = do

write2log "AAA"

write2log "BBB"

return 42

должен дать такой результат:

GHCi> runLogg logTst'

Logged "AAABBB" 42

А тест (подразумевающий импорт Control.Monad.Trans.State и Control.Monad.Trans.Class)

stLog :: StateT Integer Logg Integer

stLog = do

modify (+1)

a <- get

lift $ write2log $ show $ a \* 10

put 42

return $ a \* 100

— такой:

GHCi> runLogg $ runStateT stLog 2

Logged "30" (300,42)

Ответ:

write2log :: Monad m => String -> LoggT m ()

write2log s = LoggT $ return $ Logged s ()

type Logg = LoggT Identity

runLogg :: Logg a -> Logged a

runLogg = runIdentity . runLoggT

### Шаг 12:

В последнем примере предыдущей задачи функция lift :: (MonadTrans t, Monad m) => m a -> t m a позволяла поднять вычисление из внутренней монады (в примере это был Logg) во внешний трансформер (StateT Integer). Это возможно, поскольку для трансформера StateT s реализован представитель класса типов MonadTrans из Control.Monad.Trans.Class.

Сделайте трансформер LoggT представителем этого класса MonadTrans, так чтобы можно было поднимать вычисления из произвольной внутренней монады в наш трансформер:

instance MonadTrans LoggT where

lift = undefined

logSt :: LoggT (State Integer) Integer

logSt = do

lift $ modify (+1)

a <- lift get

write2log $ show $ a \* 10

lift $ put 42

return $ a \* 100

Проверка:

GHCi> runState (runLoggT logSt) 2

(Logged "30" 300,42)

Ответ:

instance MonadTrans LoggT where

{-

(>>=):: m a -> t m a

m a -> LoggT (m a)

-}

--lift m = LoggT (m>>=(\q->return $ Logged "" q))

--lift m = LoggT $ m >>= return . Logged mempty

lift = LoggT . (=<<) (return . Logged mempty)

## Трансформер StateT

### Шаг 4:

Реализуйте функции evalStateT и execStateT.

Ответ:

evalStateT :: Monad m => StateT s m a -> s -> m a

evalStateT st s = fst <$> runStateT st s

execStateT :: Monad m => StateT s m a -> s -> m s

execStateT st s = snd <$> runStateT st s

### Шаг 5:

Нетрудно понять, что монада State более «сильна», чем монада Reader: вторая тоже, в некотором смысле, предоставляет доступ к глобальному состоянию, но только, в отличие от первой, не позволяет его менять. Покажите, как с помощью StateT можно эмулировать ReaderT:

GHCi> evalStateT (readerToStateT $ asks (+2)) 4

6

GHCi> runStateT (readerToStateT $ asks (+2)) 4

(6,4)

Ответ:

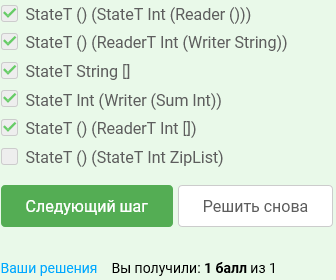
readerToStateT :: Monad m => ReaderT r m a -> StateT r m a

readerToStateT r = StateT $ \s-> flip (,) s <$> runReaderT r s

### Шаг 9:

Какие из перечисленных конструкторов типов являются представителями класса Applicative? (*Мы пока ещё не видели реализацию представителя класса* Monad *для нашего трансформера* StateT*, но предполагается, что все уже догадались, что она скоро появится.*)

Ответ:



### Шаг 11:

Неудачное сопоставление с образцом для реализованного на предыдущих видео-степах трансформера StateT аварийно прерывает вычисление:

GHCi> sl2 = StateT $ \st -> [(st,st),(st+1,st-1)]

GHCi> runStateT (do {6 <- sl2; return ()}) 5

\*\*\* Exception: Pattern match failure in do expression ...

Исправьте реализацию таким образом, чтобы обработка такой ситуации переадресовывалась бы внутренней монаде:

GHCi> sl2 = StateT $ \st -> [(st,st),(st+1,st-1)]

GHCi> runStateT (do {6 <- sl2; return ()}) 5

[((),4)]

GHCi> sm = StateT $ \st -> Just (st+1,st-1)

GHCi> runStateT (do {42 <- sm; return ()}) 5

Nothing

Ответ:

--import Control.Monad(ap) Особенности проверяющий системы

--ap:: Monad m => m (a -> b) -> m a -> m b

ap f v = f >>= flip (<$>) v

newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }

state :: Monad m => (s -> (a, s)) -> StateT s m a

state f = StateT (return . f)

execStateT :: Monad m => StateT s m a -> s -> m s

execStateT m = fmap snd . runStateT m

evalStateT :: Monad m => StateT s m a -> s -> m a

evalStateT m = fmap fst . runStateT m

instance Functor m => Functor (StateT s m) where

fmap f m = StateT $ \st -> fmap updater $ runStateT m st

where updater ~(x, s) = (f x, s)

instance Monad m => Applicative (StateT s m) where

pure = return

(<\*>) = ap

instance Monad m => Monad (StateT s m) where

return x = StateT $ \ s -> return (x, s)

m >>= k = StateT $ \s -> do

~(x, s') <- runStateT m s

runStateT (k x) s'

fail = StateT . const . fail

--fail msg = StateT $ \\_ -> fail msg

### Шаг 14:

Те из вас, кто проходил первую часть нашего курса, конечно же помнят, последнюю задачу из него. В тот раз всё закончилось монадой State, но сейчас с неё все только начинается!

data Tree a = Leaf a | Fork (Tree a) a (Tree a)

Вам дано значение типа Tree (), иными словами, вам задана *форма* дерева. От вас требуется сделать две вещи: во-первых, пронумеровать вершины дерева, обойдя их *in-order* обходом (левое поддерево, вершина, правое поддерево); во-вторых, подсчитать количество листьев в дереве.

GHCi> numberAndCount (Leaf ())

(Leaf 1,1)

GHCi> numberAndCount (Fork (Leaf ()) () (Leaf ()))

(Fork (Leaf 1) 2 (Leaf 3),2)

Конечно, можно решить две подзадачи по-отдельности, но мы сделаем это всё за один проход. Если бы вы писали решение на императивном языке, вы бы обошли дерево, поддерживая в одной переменной следующий доступный номер для очередной вершины, а в другой — количество встреченных листьев, причем само значение второй переменной, по сути, в процессе обхода не требуется. Значит, вполне естественным решением будет завести состояние для первой переменной, а количество листьев накапливать в «логе»-моноиде.

Вот так выглядит код, запускающий наше вычисление и извлекающий результат:

numberAndCount :: Tree () -> (Tree Integer, Integer)

numberAndCount t = getSum <$> runWriter (evalStateT (go t) 1)

where

go :: Tree () -> StateT Integer (Writer (Sum Integer)) (Tree Integer)

go = undefined

Вам осталось только описать само вычисление — функцию go.

Ответ:

{-import Control.Monad.Trans.Writer

import Control.Monad.Trans.State

import Control.Monad.Trans.Class(lift)

import Data.Monoid (Sum(..))

data Tree a = Leaf a | Fork (Tree a) a (Tree a) deriving Show

u = ()

wkTree = Fork (Fork (Leaf u) u (Fork (Leaf u) u (Leaf u))) u (Fork (Leaf u) u (Fork (Leaf u) u (Leaf u)))

{-

f

b g

a d h j

c e i k

-}

numberAndCount :: Tree () -> (Tree Integer, Integer)

numberAndCount t = getSum <$> runWriter (evalStateT (go t) 1)-}

go :: Tree () -> StateT Integer (Writer (Sum Integer)) (Tree Integer)

go (Fork l \_ r) = do

l'<-go l

s<-get

modify succ --put $ s+1

r'<-go r

return $ Fork l' s r'

go (Leaf \_) = do

lift $ tell 1

s<-get

modify succ --put $ s+1

return $ Leaf s

--main = print $ numberAndCount wkTree

## Трансформер ExceptT

### Шаг 6:

Представьте, что друг принес вам игру. В этой игре герой ходит по полю. За один ход он может переместиться на одну клетку вверх, вниз, влево и вправо (стоять на месте нельзя). На поле его поджидают различные опасности, такие как пропасти (chasm) и ядовитые змеи (snake). Если игрок наступает на клетку с пропастью или со змеёй, он умирает.

data Tile = Floor | Chasm | Snake

deriving Show

data DeathReason = Fallen | Poisoned

deriving (Eq, Show)

Карта задается функцией, отображающей координаты клетки в тип этой самой клетки:

type Point = (Integer, Integer)

type GameMap = Point -> Tile

Ваша задача состоит в том, чтобы реализовать функцию

moves :: GameMap -> Int -> Point -> [Either DeathReason Point]

принимающую карту, количество шагов и начальную точку, а возвращающую список всех возможных исходов (с повторениями), если игрок сделает заданное число шагов из заданной точки. Заодно реализуйте функцию

waysToDie :: DeathReason -> GameMap -> Int -> Point -> Int

показывающую, сколькими способами игрок может умереть данным способом, сделав заданное число шагов из заданной точки.

Например, для такого поля:

map1 :: GameMap

map1 (2, 2) = Snake

map1 (4, 1) = Snake

map1 (x, y)

| 0 < x && x < 5 && 0 < y && y < 5 = Floor

| otherwise = Chasm

| 0 1 2 3 4 5

--------------

0| o o o o o o

1| o s o

2| o s o

3| o o

4| o o

5| o o o o o o

ожидаются такие ответы:

GHCi> waysToDie Poisoned map1 1 (4,2)

1 -- можно пойти к змее наверх

GHCi> waysToDie Poisoned map1 2 (4,2)

2 -- можно пойти к змее наверх или к змее влево

GHCi> waysToDie Poisoned map1 3 (4,2)

5 -- за три шага к левой змее, по-прежнему можно дойти одним способом,

-- а к правой — уже четырьмя (вверх, влево-вверх-вправо,

-- влево-вправо-вверх, вниз-вверх-вверх)

GHCi> waysToDie Poisoned map1 4 (4,2)

13

Гарантируется, что изначально игрок стоит на пустой клетке.

*Подсказка: не забывайте, в каком уроке эта задача.*

Ответ:

{-import Control.Monad.Trans.Except

import Control.Monad.Trans.Class(lift)

data Tile = Floor | Chasm | Snake

deriving Show

data DeathReason = Fallen | Poisoned

deriving (Eq, Show)

type Point = (Integer, Integer)

type GameMap = Point -> Tile-}

moves :: GameMap -> Int -> Point -> [Either DeathReason Point]

moves f = (runExceptT .) . comp

where

comp :: Int -> Point -> ExceptT DeathReason [] Point

comp 0 p = return p

comp m (x,y) = do

next<-lift [(x+1,y),(x-1,y),(x,y+1),(x,y-1)]

case f next of

Chasm -> throwE Fallen

Snake -> throwE Poisoned

\_ -> lift $ return () --continue calculations

comp (m-1) next

lift = ExceptT . fmap Right -- Функция добавлена из-за особенностей проверяющий системы

throwE = ExceptT . return . Left -- Функция добавлена из-за особенностей проверяющий системы

waysToDie :: DeathReason -> GameMap -> Int -> Point -> Int

waysToDie d f m p = length $ filter (either (==d) (const False)) $ moves f m p

{-{-

| 0 1 2 3 4 5

--------------

0| o o o o o o

1| o x s o

2| o s . o

3| o x o

4| o x o

5| o o o o o o

1)3 3 3 1 3 1 4 4 4 2 4 2

Pion Fall Fall Pion

-}

map1 :: GameMap

map1 (2, 2) = Snake

map1 (4, 1) = Snake

map1 (x, y)

| 0 < x && x < 5 && 0 < y && y < 5 = Floor

| otherwise = Chasm

main = print $ moves map1 4 (4,2)-}

### Шаг 8:

Следующий код

import Control.Monad.Trans.Maybe

import Data.Char (isNumber, isPunctuation)

askPassword0 :: MaybeT IO ()

askPassword0 = do

liftIO $ putStrLn "Enter your new password:"

value <- msum $ repeat getValidPassword0

liftIO $ putStrLn "Storing in database..."

getValidPassword0 :: MaybeT IO String

getValidPassword0 = do

s <- liftIO getLine

guard (isValid0 s)

return s

isValid0 :: String -> Bool

isValid0 s = length s >= 8

&& any isNumber s

&& any isPunctuation s

используя трансформер MaybeT и свойства функции msum, отвергает ввод пользовательского пароля, до тех пор пока он не станет удовлетворять заданным критериям. Это можно проверить, вызывая его в интерпретаторе

GHCi> runMaybeT askPassword0

Используя пользовательский тип ошибки и трансформер ExceptT вместо MaybeT, модифицируйте приведенный выше код так, чтобы он выдавал пользователю сообщение о причине, по которой пароль отвергнут.

data PwdError = PwdError String

type PwdErrorIOMonad = ExceptT PwdError IO

askPassword :: PwdErrorIOMonad ()

askPassword = do

liftIO $ putStrLn "Enter your new password:"

value <- msum $ repeat getValidPassword

liftIO $ putStrLn "Storing in database..."

getValidPassword :: PwdErrorIOMonad String

getValidPassword = undefined

Ожидаемое поведение:

GHCi> runExceptT askPassword

Enter your new password:

qwerty

Incorrect input: password is too short!

qwertyuiop

Incorrect input: password must contain some digits!

qwertyuiop123

Incorrect input: password must contain some punctuation!

qwertyuiop123!!!

Storing in database...

GHCi>

Ответ:

import Control.Monad.Trans.Except

import Control.Monad.IO.Class (liftIO)

import Data.Foldable (msum)

import Data.Char (isNumber, isPunctuation)

import Data.Monoid((<>))

{- Не снимайте комментарий - эти объявления даны в вызывающем коде

newtype PwdError = PwdError String

type PwdErrorIOMonad = ExceptT PwdError IO

askPassword :: PwdErrorIOMonad ()

askPassword = do

liftIO $ putStrLn "Enter your new password:"

value <- msum $ repeat getValidPassword

liftIO $ putStrLn "Storing in database..."

-}

instance Monoid PwdError where

mempty = PwdError mempty

mappend (PwdError l) (PwdError r) = PwdError $ l <> r

getValidPassword :: PwdErrorIOMonad String

getValidPassword = do

s <- liftIO getLine

isValid s `catchE` (\p@(PwdError err)-> liftIO (putStrLn $ "Incorrect input: " <> err) >> throwE p)

return s

isValid :: String -> PwdErrorIOMonad ()

isValid s | length s < 8 = throwE $ PwdError "password is too short!"

| not (any isNumber s) = throwE $ PwdError "password must contain some digits!"

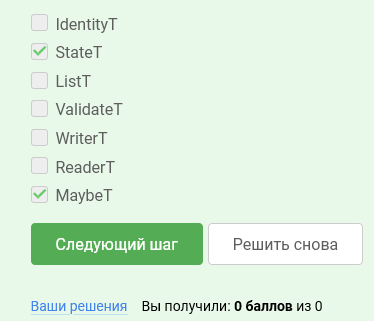
| not (any isPunctuation s) = throwE $ PwdError "password must contain some punctuation!"

| otherwise = return ()

### Шаг 11:

Попробуйте предположить, каким ещё трансформерам для реализации правильного представителя класса Applicative не достаточно, чтобы внутренний контейнер был лишь аппликативным функтором, а нужна полноценная монада?

Ответ:



### Шаг 12:

Вспомним функцию tryRead:

data ReadError = EmptyInput | NoParse String

deriving Show

tryRead :: Read a => String -> Except ReadError a

Измените её так, чтобы она работала в трансформере ExceptT.

Ответ:

--import Control.Monad.Trans.Except

import Text.Read(readMaybe)

{-data ReadError = EmptyInput | NoParse String

deriving Show-}

tryRead :: (Read a, Monad m) => String -> ExceptT ReadError m a

tryRead s | null s = throwE EmptyInput

| otherwise = maybe (throwE $ NoParse s) return $ readMaybe s

### Шаг 13:

С деревом мы недавно встречались:

data Tree a = Leaf a | Fork (Tree a) a (Tree a)

Вам на вход дано дерево, содержащее целые числа, записанные в виде строк. Ваша задача обойти дерево *in-order* (левое поддерево, вершина, правое поддерево) и просуммировать числа до первой строки, которую не удаётся разобрать функцией tryRead из прошлого задания (или до конца дерева, если ошибок нет). Если ошибка произошла, её тоже надо вернуть.

Обходить деревья мы уже умеем, так что от вас требуется только функция go, подходящая для такого вызова:

treeSum t = let (err, s) = runWriter . runExceptT $ traverse\_ go t

in (maybeErr err, getSum s)

where

maybeErr :: Either ReadError () -> Maybe ReadError

maybeErr = either Just (const Nothing)

GHCi> treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "oops")) "15" (Leaf "16")

(Just (NoParse "oops"),3)

GHCi> treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "0")) "15" (Leaf "16")

(Nothing,34)

Ответ:

{-import Control.Monad.Trans.Writer

import Control.Monad.Trans.Class(lift)

import Data.Monoid (Sum(..))

import Text.Read(readMaybe)

import Control.Monad.Trans.Except

import Data.Traversable

import Data.Foldable

data Tree a = Leaf a | Fork (Tree a) a (Tree a) deriving Show

instance Foldable Tree where

foldMap = foldMapDefault

instance Functor Tree where

fmap = fmapDefault

instance Traversable Tree where

traverse f (Fork l v r) = Fork <$> traverse f l <\*> f v <\*> traverse f r

traverse f (Leaf v) = Leaf <$> f v

treeSum t = let (err, s) = runWriter . runExceptT $ traverse\_ go t

in (maybeErr err, getSum s)

where

maybeErr :: Either ReadError () -> Maybe ReadError

maybeErr = either Just (const Nothing)

-}

go :: String -> ExceptT ReadError (Writer (Sum Integer)) ()

go s = do

v<-tryRead s

lift $ tell $ Sum v

{-main = print $ treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "0")) "15" (Leaf "16")

data ReadError = EmptyInput | NoParse String

deriving Show

tryRead :: (Read a, Monad m) => String -> ExceptT ReadError m a

tryRead s | null s = throwE EmptyInput

| otherwise = maybe (throwE $ NoParse s) return $ readMaybe-}

## Неявный лифтинг

### Шаг 5:

Предположим мы хотим реализовать следующую облегченную версию функтора, используя многопараметрические классы типов:

class Functor' c e where

fmap' :: (e -> e) -> c -> c

Добавьте в определение этого класса типов необходимые функциональные зависимости и реализуйте его представителей для списка и Maybe так, чтобы обеспечить работоспособность следующих вызовов

GHCi> fmap' succ "ABC"

"BCD"

GHCi> fmap' (^2) (Just 42)

Just 1764

Ответ:

--{-# LANGUAGE MultiParamTypeClasses #-}

{-# LANGUAGE FunctionalDependencies #-}

{-# LANGUAGE FlexibleInstances #-}

class Functor' c e | c -> e where

fmap' :: (e -> e) -> c -> c

instance Functor' (Maybe e) e where

fmap' = fmap

instance Functor' [e] e where

fmap' = fmap

### Шаг 10:

В этой и следующих задачах мы продолжаем работу с трансформером LoggT [разработанным на первом уроке этой недели](https://stepik.org/lesson/38578/step/12):

data Logged a = Logged String a deriving (Eq,Show)

newtype LoggT m a = LoggT { runLoggT :: m (Logged a) }

write2log :: Monad m => String -> LoggT m ()

type Logg = LoggT Identity

runLogg :: Logg a -> Logged a

Теперь мы хотим сделать этот трансформер mtl-совместимым.

Избавьтесь от необходимости ручного подъема операций вложенной монады State, сделав трансформер LoggT, примененный к монаде с интерфейсом MonadState, представителем этого (MonadState) класса типов:

instance MonadState s m => MonadState s (LoggT m) where

get = undefined

put = undefined

state = undefined

logSt' :: LoggT (State Integer) Integer

logSt' = do

modify (+1) -- no lift!

a <- get -- no lift!

write2log $ show $ a \* 10

put 42 -- no lift!

return $ a \* 100

Проверка:

GHCi> runState (runLoggT logSt') 2

(Logged "30" 300,42)

Ответ:

{-# LANGUAGE MultiParamTypeClasses #-}

{-# LANGUAGE FlexibleInstances #-}

{-# LANGUAGE UndecidableInstances #-}

import Data.Functor.Identity

import Control.Monad.State

instance MonadState s m => MonadState s (LoggT m) where

state = lift . state

### Шаг 11:

Избавьтесь от необходимости ручного подъема операций вложенной монады Reader, сделав трансформер LoggT, примененный к монаде с интерфейсом MonadReader, представителем этого (MonadReader) класса типов:

instance MonadReader r m => MonadReader r (LoggT m) where

ask = undefined

local = undefined

reader = undefined

Для упрощения реализации функции local имеет смысл использовать вспомогательную функцию, поднимающую стрелку между двумя «внутренними представлениями» трансформера LoggT в стрелку между двумя LoggT:

mapLoggT :: (m (Logged a) -> n (Logged b)) -> LoggT m a -> LoggT n b

mapLoggT f = undefined

Тест:

logRdr :: LoggT (Reader [(Int,String)]) ()

logRdr = do

Just x <- asks $ lookup 2 -- no lift!

write2log x

Just y <- local ((3,"Jim"):) $ asks $ lookup 3 -- no lift!

write2log y

Ожидаемый результат:

GHCi> runReader (runLoggT logRdr) [(1,"John"),(2,"Jane")]

Logged "JaneJim" ()

Ответ:

{-# LANGUAGE MultiParamTypeClasses #-}

{-# LANGUAGE FlexibleInstances #-}

{-# LANGUAGE UndecidableInstances #-}

import Data.Functor.Identity

import Control.Monad.Reader

mapLoggT :: (m (Logged a) -> n (Logged b)) -> LoggT m a -> LoggT n b

mapLoggT f v = LoggT $ f $ runLoggT v

instance MonadReader r m => MonadReader r (LoggT m) where

ask = lift ask

{-

local :: (r -> r) -> m a -> m a

(r -> r) -> LoggT m a -> LoggT m a

-}

--local f v = mapLoggT (local f) v

local = mapLoggT . local

### Шаг 12:

Чтобы избавится от необходимости ручного подъема операции write2log, обеспечивающей стандартный интерфейс вложенного трансформера LoggT, можно поступить по аналогии с другими трансформерами библиотеки mtl. А именно, разработать класс типов MonadLogg, выставляющий этот стандартный интерфейс

class Monad m => MonadLogg m where

w2log :: String -> m ()

logg :: Logged a -> m a

(Замечание: Мы переименовываем функцию write2log в w2log, поскольку хотим держать всю реализацию в одном файле исходного кода. При следовании принятой в библиотеках transformers/mtl идеологии они имели бы одно и то же имя, но были бы определены в разных модулях. При работе с transformers мы импортировали бы свободную функцию c квалифицированным именем Control.Monad.Trans.Logg.write2log, а при использовании mtl работали бы с методом класса типов MonadLogg с полным именем Control.Monad.Logg.write2log. )

Этот интерфейс, во-первых, должен выставлять сам трансформер LoggT, обернутый вокруг произвольной монады:

instance Monad m => MonadLogg (LoggT m) where

w2log = undefined

logg = undefined

Реализуйте этого представителя, для проверки используйте:

logSt'' :: LoggT (State Integer) Integer

logSt'' = do

x <- logg $ Logged "BEGIN " 1

modify (+x)

a <- get

w2log $ show $ a \* 10

put 42

w2log " END"

return $ a \* 100

Результат должен быть таким:

GHCi> runState (runLoggT logSt'') 2

(Logged "BEGIN 30 END" 300,42)

Во-вторых, интерфейс MonadLogg должен выставлять любой стандартный трансформер, обернутый вокруг монады, выставляющей этот интерфейс:

instance MonadLogg m => MonadLogg (StateT s m) where

w2log = undefined

logg = undefined

instance MonadLogg m => MonadLogg (ReaderT r m) where

w2log = undefined

logg = undefined

-- etc...

Реализуйте двух этих представителей, для проверки используйте:

rdrStLog :: ReaderT Integer (StateT Integer Logg) Integer

rdrStLog = do

x <- logg $ Logged "BEGIN " 1

y <- ask

modify (+ (x+y))

a <- get

w2log $ show $ a \* 10

put 42

w2log " END"

return $ a \* 100

Результат должен быть таким:

GHCi> runLogg $ runStateT (runReaderT rdrStLog 4) 2

Logged "BEGIN 70 END" (700,42)

Ответ:

{-# LANGUAGE MultiParamTypeClasses #-}

{-# LANGUAGE FlexibleInstances #-}

{-# LANGUAGE UndecidableInstances #-}

import Data.Functor.Identity

import Control.Monad.State

import Control.Monad.Reader

class Monad m => MonadLogg m where

w2log :: String -> m ()

logg :: Logged a -> m a

instance Monad m => MonadLogg (LoggT m) where

w2log = write2log

logg = LoggT . return

instance MonadLogg m => MonadLogg (StateT s m) where

w2log = lift . w2log

logg = lift . logg

instance MonadLogg m => MonadLogg (ReaderT r m) where

w2log = lift . w2log

logg = lift . logg

{-for ghc 8.0.2

{-# LANGUAGE FlexibleInstances #-}

{-# LANGUAGE FunctionalDependencies #-}

{-# LANGUAGE MultiParamTypeClasses #-}

{-# LANGUAGE UndecidableInstances #-}

import Control.Monad.Trans.Class (lift,MonadTrans(..))

import Control.Monad

import Data.Monoid((<>))

import Data.Functor.Identity

import Control.Applicative(liftA2)

import Control.Monad.Trans.State

import Control.Monad.Trans.Reader

write2log :: Monad m => String -> LoggT m ()

write2log s = LoggT $ return $ Logged s ()

type Logg = LoggT Identity

runLogg :: Logg a -> Logged a

runLogg = runIdentity . runLoggT

instance Functor m => Functor (LoggT m) where

fmap f v = LoggT $ fmap updater (runLoggT v)

where updater ~(Logged s v') = Logged s $ f v'

instance Applicative m => Applicative (LoggT m) where

pure v = LoggT $ pure $ Logged mempty v

f <\*> v = LoggT $ liftA2 updater (runLoggT f) (runLoggT v)

where updater ~(Logged s f) ~(Logged s' a) = Logged (s <> s') (f a)

instance Monad m => Monad (LoggT m) where

v >>= f = LoggT $ do

~(Logged s a) <- runLoggT v

~(Logged s' a') <- runLoggT $ f a

return $ Logged (s<>s') a'

fail = LoggT . fail

instance MonadTrans LoggT where

lift = LoggT . (=<<) (return . Logged mempty)

data Logged a = Logged String a deriving (Eq,Show)

newtype LoggT m a = LoggT { runLoggT :: m (Logged a) }

class Monad m => MonadLogg m where

w2log :: String -> m ()

logg :: Logged a -> m a

instance Monad m => MonadLogg (LoggT m) where

w2log = write2log

--logg :: Logged a -> (LoggT m) a

logg = LoggT . return

instance MonadLogg m => MonadLogg (StateT s m) where

w2log = lift . w2log

logg = lift . logg

instance MonadLogg m => MonadLogg (ReaderT r m) where

w2log = lift . w2log

logg = lift . logg

main = print $ ""-}

## Задачи на трансформеры

### Шаг 2:

Функция tryRead обладает единственным эффектом: в случае ошибки она должна прерывать вычисление. Это значит, что её можно использовать в любой монаде, предоставляющей возможность завершать вычисление с ошибкой, но сейчас это не так, поскольку её тип это делать не позволяет:

data ReadError = EmptyInput | NoParse String

deriving Show

tryRead :: (Read a, Monad m) => String -> ExceptT ReadError m a

Измените её так, чтобы она работала в любой монаде, позволяющей сообщать об исключительных ситуациях типа ReadError. Для этого к трансформеру ExceptT в библиотеке mtl прилагается класс типов MonadError (обратите внимание на название класса — это так сделали специально, чтобы всех запутать), находящийся в модуле Control.Monad.Except.

Ответ:

{-# LANGUAGE FlexibleContexts #-}

import Control.Monad.Except

import Text.Read(readMaybe)

--data ReadError = EmptyInput | NoParse String deriving Show

tryRead :: (Read a, MonadError ReadError m) => String -> m a

tryRead "" = throwError EmptyInput

tryRead s = maybe (throwError $ NoParse s) return $ readMaybe s

### Шаг 3:

В очередной раз у вас есть дерево строковых представлений чисел:

data Tree a = Leaf a | Fork (Tree a) a (Tree a)

и функция tryRead:

data ReadError = EmptyInput | NoParse String

deriving Show

tryRead :: (Read a, MonadError ReadError m) => String -> m a

Просуммируйте числа в дереве, а если хотя бы одно прочитать не удалось, верните ошибку:

GHCi> treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "oops")) "15" (Leaf "16")

Left (NoParse "oops")

GHCi> treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "0")) "15" (Leaf "16")

Right 34

﻿*Подумайте, что общего у этой задачи с похожей, которую вы решали ранее? В чем отличия? При чем здесь библиотека mtl?*

Ответ:

import Control.Applicative(liftA2)

treeSum :: Tree String -> Either ReadError Integer

--treeSum tr = foldr (\s ls-> liftA2 (+) (tryRead s) ls) (Right 0) tr

treeSum = foldr (liftA2 (+) . tryRead) (Right 0)

{-

{-# LANGUAGE FlexibleContexts #-}

import Control.Monad.Except

import Text.Read(readMaybe)

import Data.Traversable

import Data.Foldable

import Control.Applicative(liftA2)

data Tree a = Leaf a | Fork (Tree a) a (Tree a) deriving Show

instance Foldable Tree where

foldMap = foldMapDefault

instance Functor Tree where

fmap = fmapDefault

instance Traversable Tree where

traverse f (Fork l v r) = Fork <$> traverse f l <\*> f v <\*> traverse f r

traverse f (Leaf v) = Leaf <$> f v

data ReadError = EmptyInput | NoParse String deriving Show

tryRead :: (Read a, MonadError ReadError m) => String -> m a

tryRead "" = throwError EmptyInput

tryRead s = maybe (throwError $ NoParse s) return $ readMaybe s

treeSum :: Tree String -> Either ReadError Integer

--treeSum tr = foldr (\s ls-> liftA2 (+) (tryRead s) ls) (Right 0) tr

treeSum = foldr (liftA2 (+) . tryRead) (Right 0)

main = print$ treeSum $ Fork (Fork (Leaf "1") "2" (Leaf "9")) "15" (Leaf "16")

-}

### Шаг 4:

Вам дан список вычислений с состоянием (State s a) и начальное состояние. Требуется выполнить все эти вычисления по очереди (очередное вычисление получает на вход состояние, оставшееся от предыдущего) и вернуть список результатов. Но это ещё не всё. Ещё вам дан предикат, определяющий, разрешено некоторое состояние или нет; после выполнения очередного вычисления вы должны с помощью этого предиката проверить текущее состояние, и, если оно не разрешено, завершить вычисление, указав номер вычисления, которое его испортило.

При этом, завершаясь с ошибкой, мы можем как сохранить накопленное до текущего момента состояние, так и выкинуть его. В первом случае наша функция будет иметь такой тип:

runLimited1 :: (s -> Bool) -> [State s a] -> s -> (Either Int [a], s)

Во втором — такой:

runLimited2 :: (s -> Bool) -> [State s a] -> s -> Either Int ([a], s)

Пример:

GHCi> runLimited1 (< 3) [modify (+1), modify (+1), modify (+1), modify (+1)] 0

(Left 2,3)

GHCi> runLimited2 (< 3) [modify (+1), modify (+1), modify (+1), modify (+1)] 0

Left 2

— после выполнения вычисления с индексом 2 (нумерация с нуля) в состоянии оказалась тройка, что плохо.

GHCi> runLimited1 (< 100) [modify (+1), modify (+1), modify (+1), modify (+1)] 0

(Right [(),(),(),()],4)

GHCi> runLimited2 (< 100) [modify (+1), modify (+1), modify (+1), modify (+1)] 0

Right ([(),(),(),()],4)

— всё хорошо, предикат всё время выполнялся. Вычисления ничего не возвращали (имели тип State Int ()), так что списки получились такие неинтересные.

Если, прочитав про список, с каждым элементом которого надо совершить какое-то действие, вы сразу же подумали про traverse, то правильно сделали. Вопрос только в том, какое именно действие надо совершить с каждым элементом списка, но тут тоже всё довольно просто: надо выполнить вычисление с состоянием, описываемое элементом списка, а затем проверить состояние и, в случае проблемы, кинуть исключение:

limited p fs = traverse limit1 (zip [0..] fs)

where

limit1 (i, f) = do

a <- state (runState f)

stateIsBad <- gets (not . p)

when stateIsBad $ throwError i

pure a

(Прежде чем проходить по списку, мы ещё пронумеровали его элементы.)

Собственно, остался сущий пустяк — запустить наше новое вычисление.

runLimited1 :: (s -> Bool) -> [State s a] -> s -> (Either Int [a], s)

runLimited1 p fs s = run1 (limited p fs) s

runLimited2 :: (s -> Bool) -> [State s a] -> s -> Either Int ([a], s)

runLimited2 p fs s = run2 (limited p fs) s

А теперь задание. Реализуйте функции run1 и run2.

Ответ:

{-# LANGUAGE FlexibleContexts #-}

import Control.Monad.Except

import Control.Monad.State

import Data.Functor.Identity(runIdentity)

{-limited ::(MonadError Int m, MonadState s m) =>

(s -> Bool) -> [State s a] -> m [a]

limited p fs = traverse limit1 (zip [0..] fs)

where

limit1 (i, f) = do

a <- state (runState f)

stateIsBad <- gets (not . p)

when stateIsBad $ throwError i

pure a

runLimited1 :: (s -> Bool) -> [State s a] -> s -> (Either Int [a], s)

runLimited1 p fs s = run1 (limited p fs) s

runLimited2 :: (s -> Bool) -> [State s a] -> s -> Either Int ([a], s)

runLimited2 p fs s = run2 (limited p fs) s-}

run1 :: ExceptT Int (State s) [a] -> s -> (Either Int [a], s)

run1 = (runIdentity .) . runStateT . runExceptT

--run1 m s = runIdentity (runStateT (runExceptT m) s)

run2 :: StateT s (Either Int) [a] -> s -> Either Int ([a], s)

run2 = runStateT

{-

Решение поставленной задачи будет состоять из трех этапов:

1) Подготовительный этап

2) Поиск сигнатуры для функции limited

3) Реализация run1 run2

--------ЭТАП 1(Подготовительный)-------

В рассматриваемой задачи приводится несколько связанных листингов исходного кода.

Их совмещение в единую область позволит упростить возможную трудности восприятия.

Выполним это:

limited p fs = traverse limit1 (zip [0..] fs)

where

limit1 (i, f) = do

a <- state (runState f)

stateIsBad <- gets (not . p)

when stateIsBad $ throwError i

pure a

runLimited1 :: (s -> Bool) -> [State s a] -> s -> (Either Int [a], s)

runLimited1 p fs s = run1 (limited p fs) s

runLimited2 :: (s -> Bool) -> [State s a] -> s -> Either Int ([a], s)

runLimited2 p fs s = run2 (limited p fs) s

run1 = undefined

run2 = undefined

Попытка компиляции будет провальной, так как однозначный вывод типов невозможен.

Помимо этого, реализация функций run1 и run2 требует детального

понимания происходящего, в связи с этим,

оптимальное продолжения решения задачи - вывод сигнатуры типов для функции limited.

--------ЭТАП 2(Поиск сигнатуры для функции limited)-------

1. limited вызывается функцией runLimited1 с двумя аргументами - предикат и список

состояний, следовательно, первое приближение искомой сигнатуры имеет вид:

limited :: (s -> Bool) -> [State s a] -> ?

2. Тело рассматриваемой функции позволяет утверждать эквиволентность между

возвращаемым типом функции limited и traverse. Поэтому

дальнейший анализ будет посвящен сигнатуре traverse

3. С точки зрения типов, traverse характеризуется так:

Applicative f => (a -> f b) -> t a -> f (t b)

4. В данном случае, первый аргумент traverse равен функции limit. Второй же базируется на

"почленном смешивании" двух списков (zip [0..] fs), что в конечном итоге порождает тип:

(zip [0..] fs) :: [(Int, State s a)].

Внесём полученную информацию в сигнатуру из шага №3:

a) traverse :: Applicative f => (a -> f b) -> [(Int, State s a)] -> f [b]

b) traverse :: Applicative f => ((Int, State s a) -> f b) -> [(Int, State s a)] -> f [b]

5. Вполне очевидно, что возвращаемый тип traverse находится в прямой зависимости

от функции-аргумента limit1, поэтому, несколько следующих шагов

будут направлены на эту функцию(limit1)

6. Левое определение limit1 гласит о единичном количестве входящих аргументов,

более того, частично рассчитанная сигнатура из шага 4(последняя) форсирует

тип этого аргумента, он выражен парой (Int, State s a). Сформируем

подпись типов для limit:

limit1 :: (Int, State s a) -> ?

7. Данный этап приближается к завершению. Все сигнатуры этого этапа будут

замкнуты(закончены), как только возвращаемый тип limit1 окажется известен.

8. Обозревая реализацию функции limit1 заметим следующие:

1) Её воплощение полностью описано do-нотацией, значит возвращаемый тип является монадой.

2) Важнейшим наблюдением выступает наличие функции throwError.

Что однозначно свидетельствует о библиотеке mtl и классе типов MonadEror.

Обнаруженный факт доказывает об отсуствии конкретный монады,

вместо этого limit1 работает с абстрактной монадой, для

который выставлен интерфейс MonadError

3) Помимо этого, в блоке do используются инструкции монады State (state, gets).

Предыдущий шаг заявляет об абстрактной монаде с интерфейсом MonadError, однако

применение инструкций state и gets вводит интерфейс MonadState

9. Пользуясь извлеченный информацией из шага №8 окончим сигнатуру limit1:

1) limit1 :: (Int, State s a) -> ?

2) limit1 :: (MonadError Int m, MonadState s m) => (Int, State s a) -> m a

10. Все необходимые сведения получены, замкнем сигнатуры traverse и limited:

1) traverse :: (MonadError Int m, MonadState s m) => ((Int, State s a) -> m a) -> [(Int, State s a)] -> m [a]

2) limited :: (MonadError Int m, MonadState s m) => (s -> Bool) -> [State s a] -> m [a]

--------ЭТАП 3(Реализация run1 run2)-------

TODO

-}

### Шаг 5:

Почти у каждого трансформера монад есть соответствующий ему класс типов (StateT — MonadState, ReaderT — MonadReader), хотя иногда имя класса построено иначе, нежели имя трансформера (ExceptT — MonadError).

Как называется класс, соответствующий трансформеру MaybeT? (Вопрос с небольшим подвохом.)

Ответ:

MonadPlus

### Шаг 6:

Чтобы закончить наш курс ярко, предлагаем вам с помощью этой задачи в полной мере почувствовать на себе всю мощь continuation-passing style. Чтобы успешно решить эту задачу, вам нужно хорошо понимать, как работает CPS и монада ContT (а этого, как известно, никто не понимает). *Кстати, это была подсказка*.

[Сопрограмма](https://ru.wikipedia.org/wiki/%D0%A1%D0%BE%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0) (корутина, [coroutine](https://en.wikipedia.org/wiki/Coroutine)) это обобщение понятия подпрограммы (по-простому говоря, функции). У функции, в отличие от сопрограммы, есть одна точка входа (то, откуда она начинает работать), а точек выхода может быть несколько, но выйти через них функция может только один раз за время работы; у сопрограммы же точек входа и выхода может быть несколько. Проще всего объяснить на примере:

coroutine1 = do

  tell "1"

  yield

  tell "2"

coroutine2 = do

  tell "a"

  yield

  tell "b"

GHCi> execWriter (runCoroutines coroutine1 coroutine2)

"1a2b"

Здесь используется специальное действие yield, которое передает управление другой сопрограмме. Когда другая сопрограмма возвращает управление (с помощью того же yield или завершившись), первая сопрограмма продолжает работу с того места, на котором остановилась в прошлый раз.

В общем случае, одновременно могут исполняться несколько сопрограмм, причем при передаче управления, они могут обмениваться значениями. В этой задаче достаточно реализовать сопрограммы в упрощенном виде: одновременно работают ровно две сопрограммы и значениями обмениваться они не могут.

Реализуйте трансформер CoroutineT, функцию yield для передачи управления и функцию runCoroutines для запуска. Учтите, что одна сопрограмма может завершиться раньше другой; другая должна при этом продолжить работу:

coroutine3, coroutine4 :: CoroutineT (Writer String) ()

coroutine3 = do

tell "1"

yield

yield

tell "2"

coroutine4 = do

tell "a"

yield

tell "b"

yield

tell "c"

yield

tell "d"

yield

GHCi> execWriter (runCoroutines coroutine3 coroutine4)

"1ab2cd"

Ответ:

{-# LANGUAGE FlexibleInstances, UndecidableInstances, MultiParamTypeClasses #-}

{-# LANGUAGE GeneralizedNewtypeDeriving #-}

import Data.Foldable

import Control.Monad.Cont

import Control.Monad.Writer

import Control.Monad.State

newtype CoroutineT m a = CoroutineT {runCoroutineT ::

ContT () (StateT (Maybe (CoroutineT m ())) m) a }

deriving (Functor,Applicative,Monad,MonadCont,MonadIO)

getState :: Monad m => CoroutineT m (Maybe (CoroutineT m ()))

getState = CoroutineT $ lift get

putState :: Monad m => Maybe (CoroutineT m ()) -> CoroutineT m ()

putState = CoroutineT . lift . put

instance (MonadWriter w m) => MonadWriter w (CoroutineT m) where

--tell = CoroutineT . (lift . lift . tell)

tell w = CoroutineT $ lift $ lift $ tell w

listen = undefined

pass = undefined

ending ::Monad m => CoroutineT m () -> CoroutineT m ()

ending x = do

x

state <- getState

case state of

Just next -> putState Nothing >> next

Nothing -> CoroutineT $ ContT $ (\\_->return ())

runCoroutines :: Monad m => CoroutineT m () -> CoroutineT m () -> m ()

runCoroutines c c' = evalStateT (

runContT (runCoroutineT $ ending c) (\\_-> return (){-undefined-})

) (Just $ ending c')

yield :: Monad m => CoroutineT m ()

yield = callCC $ \k -> do

state <- getState

case state of

Just next -> putState (Just $ k ()) >> next

Nothing -> k ()

{-main = print $ tests

check s1 s2= if (s1==s2) then "Passed\n" else "Fail\n"

run c1 c2 s = check (execWriter $ runCoroutines c1 c2) s

tests = run a1 a2 "1a2b" ++

run b1 b2 "1a23bcg" ++

run c1 c2 "1ab2" ++

run d1 d2 "1ab234"++

run te te "12341234"

out = tell

a1 = do

out "1"

yield

out "2"

a2 = do

out "a"

yield

out "b"

b1 = do

out "1"

yield

out "2"

out "3"

b2 = do

out "a"

yield

out "b"

out "c"

yield

out "g"

c1 = do

out "1"

yield

out "2"

c2 = do

out "a"

out "b"

d1 = do

out "1"

yield

out "2"

yield

out "3"

yield

out "4"

d2 = do

out "a"

out "b"

te = do

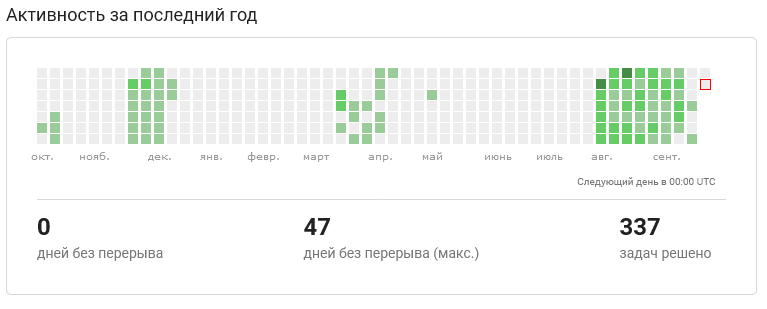
out "1"

out "2"

out "3"

out "4"-}

# ГРАФИК АКТИВНОСТИ



Начало прохождение курса: 28.08.2021.

Прохождение курса было завершено: 26.09.2021.

# СЕРТИФИКАТ

[скрыто, сертификат 100%, <https://stepik.org/course/693/syllabus>]