Как построить (символьно интерпретируемый) EDSL

Tagless final, monad transformers, RebindableSyntax и все-все-все

Григорий Волков

20 ноября 2019

ИСП РАН

Про объявление языков

Вспомним и дополним прошлый семинар

- Church encoding: соответствие между данными и функциями

Вспомним и дополним прошлый семинар

- Church encoding: соответствие между данными и функциями
- на прошлом семинаре в основном рассматривалось «от данных к функциям»

Вспомним и дополним прошлый семинар

- Church encoding: соответствие между данными и функциями
- на прошлом семинаре в основном рассматривалось «от данных к функциям»
    - есть `Either a b`, нужен только для выбора между обработчиками, почему бы сразу не вызывать эти обработчики вместо конструирования `Left`/`Right`

Вспомним и дополним прошлый семинар

- Church encoding: соответствие между данными и функциями
- на прошлом семинаре в основном рассматривалось «от данных к функциям»
  - есть `Either a b`, нужен только для выбора между обработчиками, почему бы сразу не вызывать эти обработчики вместо конструирования `Left`/`Right`
- а что если пойти «от функций к данным»?

От функций к данным

- если есть набор команд (интерфейс)

```
class Console r where
  writeLn :: String → r ()
  readLn  :: r String
```

От функций к данным

- если есть набор команд (интерфейс)

```
class Console r where
  writeLn :: String → r ()
  readLn  :: r String
```

- то его можно представить в виде данных (GADT)

```
data Console a where
  WriteLn :: String → Console ()
  ReadLn  :: Console String
```

От функций к данным

- если есть набор команд (интерфейс)

```
class Console r where
  writeLn :: String → r ()
  readLn  :: r String
```

- то его можно представить в виде данных (GADT)

```
data Console a where
  WriteLn :: String → Console ()
  ReadLn  :: Console String
```

- заглядывая вперед на пару семинаров (?): такое представление используется некоторыми системами эффектов

От функций к данным, часть 2

- если есть рекурсивный набор команд (язык)

```
class Console r where
  writeLn     :: String → r ()
  sequence    :: [r ()] → r ()
  loopForever :: r () → r ()
```

От функций к данным, часть 2

- если есть рекурсивный набор команд (язык)

```
class Console r where
  writeLn     :: String → r ()
  sequence    :: [r ()] → r ()
  loopForever :: r () → r ()
```

- то его можно представить в виде данных (GADT)

```
data Console a where
  WriteLn     :: String → Console ()
  Sequence    :: [Console ()] → Console ()
  LoopForever :: Console () → Console ()
```

От функций к данным, часть 2.1

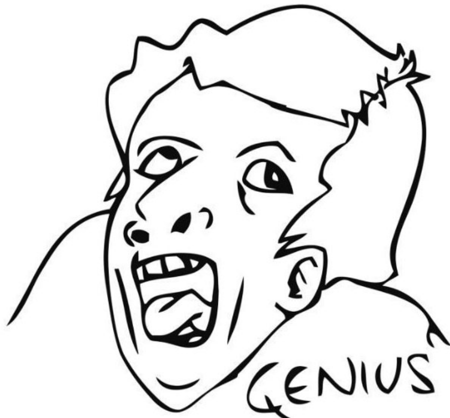
- это же синтаксическое дерево!

От функций к данным, часть 2.1

- это же синтаксическое дерево!
- пример значения: `LoopForever (Sequence [WriteLn "Hello",WriteLn "World"])`

От функций к данным, часть 2.1

- это же синтаксическое дерево!
- пример значения: `LoopForever (Sequence [WriteLn "Hello",WriteLn "World"])`
- мы установили связь между языком и синтаксическим деревом!



Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать
    - все видели примеры с арифметикой и т.д.

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать
  - все видели примеры с арифметикой и т.д.
- а хорошо как раз писать сразу в church-encoded виде!

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать
    - все видели примеры с арифметикой и т.д.
- а хорошо как раз писать сразу в church-encoded виде!
- тогда у нас четкое разделение синтаксиса (`class`) и семантики (`instance`)

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать
  - все видели примеры с арифметикой и т.д.
- а хорошо как раз писать сразу в church-encoded виде!
- тогда у нас четкое разделение синтаксиса (`class`) и семантики (`instance`)
- и все это расширяется во все стороны

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать
    - все видели примеры с арифметикой и т.д.
- а хорошо как раз писать сразу в church-encoded виде!
- тогда у нас четкое разделение синтаксиса (`class`) и семантики (`instance`)
- и все это расширяется во все стороны
    - у одного языка может быть сколько угодно интерпретаторов

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать
    - все видели примеры с арифметикой и т.д.
- а хорошо как раз писать сразу в church-encoded виде!
- тогда у нас четкое разделение синтаксиса (`class`) и семантики (`instance`)
- и все это расширяется во все стороны
    - у одного языка может быть сколько угодно интерпретаторов
    - один интерпретатор может поддерживать сколько угодно языков

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать
    - все видели примеры с арифметикой и т.д.
- а хорошо как раз писать сразу в church-encoded виде!
- тогда у нас четкое разделение синтаксиса (`class`) и семантики (`instance`)
- и все это расширяется во все стороны
    - у одного языка может быть сколько угодно интерпретаторов
    - один интерпретатор может поддерживать сколько угодно языков
        - «язык» в глобальном смысле может быть представлен как несколько языков-классов

Tagless final за один слайд

- Typed tagless final interpreters (Kiselyov 2012)
- наивный способ написать язык и его интерпретатор --- как раз объявить синтаксическое дерево и с ним работать
    - все видели примеры с арифметикой и т.д.
- а хорошо как раз писать сразу в church-encoded виде!
- тогда у нас четкое разделение синтаксиса (`class`) и семантики (`instance`)
- и все это расширяется во все стороны
    - у одного языка может быть сколько угодно интерпретаторов
    - один интерпретатор может поддерживать сколько угодно языков
        - «язык» в глобальном смысле может быть представлен как несколько языков-классов
- можно вводить новые конструкции или новые интерпретации из разных библиотек и т.д.

Tagless final: классический пример

```
class Lambda repr where
  int :: Int → repr Int
  add :: repr Int → repr Int → repr Int
  lam :: (repr a → repr b) → repr (a → b)
  app :: repr (a → b) → repr a → repr b

newtype Run a = Run { unRun :: a }

instance Lambda Run where
  int = Run
  add l r = Run $ unRun l + unRun r
  lam f = Run $ unRun . f . Run
  app l r = Run $ (unRun l) (unRun r)
```

Tagless final: классический пример (печать с `wl-pprint`)

```
class Lambda repr where
  int :: Int → repr Int
  add :: repr Int → repr Int → repr Int
  lam :: (repr a → repr b) → repr (a → b)
  app :: repr (a → b) → repr a → repr b

newtype Print a = Print { unPrint :: Int → Doc }
instance Lambda Print where
  int = Print . const . PP.int
  add (Print l) (Print r) = Print $ \c →
    l c <+> text "+" <+> r c
  lam f = Print $ \c →
    text "\\var" <> PP.int c <+> text "→" <+>
      (unPrint $ f $ Print $ const $
        text "var" <> PP.int c) (succ c)
  app (Print l) (Print r) = Print $ \c →
    parens (l c) <+> parens (r c)
```

Tagless final: классический пример (печать с wl-pprint)

```haskell
prg :: Lambda repr => repr Int
prg = (lam $ \x → x `add`
        ((lam $ \y → int 2 `add` y) `app` x)
      ) `app` (int 1 `add` int 1)

main = do putStrLn $ "Print: " <> show (unPrint prg 0)
          putStrLn $ "Run:   " <> show (unRun prg)

-- Print: (\var0 → var0 + (\var1 → 2 + var1) (var0)) (1 + 1)
-- Run:   6
```

Tagless final + эффекты

- что если мы хотим выражать последовательную композицию с зависимостью по данным (проще говоря, использовать монады)?

Tagless final + эффекты

- что если мы хотим выражать последовательную композицию с зависимостью по данным (проще говоря, использовать монады)?
- например в библиотеке `mtl` есть готовые классы..

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
-- (тут немного упрощено)
```

Tagless final + эффекты

- что если мы хотим выражать последовательную композицию с зависимостью по данным (проще говоря, использовать монады)?
- например в библиотеке `mtl` есть готовые классы..

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
-- (тут немного упрощено)
```

- программы на «языке» Monad+MonadState выглядят так:

```
prog = do
  put 123
  (+ 1) <$> get
```

Tagless final + эффекты

- что если мы хотим выражать последовательную композицию с зависимостью по данным (проще говоря, использовать монады)?
- например в библиотеке mtl есть готовые классы..

```
class Monad m => MonadState s m | m -> s where
  get :: m s
  put :: s -> m ()
-- (тут немного упрощено)
```

- программы на «языке» Monad+MonadState выглядят так:

```
prog = do
  put 123
  (+ 1) <$> get
```

- ∃ «нормальный» интерпретатор, но про него пока не будем

Tagless final + эффекты + символьное исполнение

- интересно для такого языка написать принтер

Tagless final + эффекты + символьное исполнение

- интересно для такого языка написать принтер
- реализовать `MonadState` для `Print` тривиально:

```
instance MonadState Int Print where
  get = Print $ const $ text "get"
  put x = Print $ const $ text "put" <+> PP.int x
```

Tagless final + эффекты + символьное исполнение

- интересно для такого языка написать принтер
- реализовать `MonadState` для `Print` тривиально:

```
instance MonadState Int Print where
  get = Print $ const $ text "get"
  put x = Print $ const $ text "put" <+> PP.int x
```

- а вот с `Monad` у нас проблемы!

```
instance Monad Print where
  (>>=) :: Print a → (a → Print b) → Print b
  l >>= r = Print $ \c → unPrint l c <+> text ">>="
              <+> unPrint (r ???undefined???) c
```

Tagless final + эффекты + символьное исполнение

- интересно для такого языка написать принтер
- реализовать MonadState для Print тривиально:

```
instance MonadState Int Print where
  get = Print $ const $ text "get"
  put x = Print $ const $ text "put" <+> PP.int x
```

- а вот с Monad у нас проблемы!

```
instance Monad Print where
  (>>=) :: Print a → (a → Print b) → Print b
  l >>= r = Print $ \c → unPrint l c <+> text ">>="
              <+> unPrint (r ???undefined???) c
```

- мы исполняем символьно --- нам неоткуда взять настоящее значение типа a

```
newtype Print a = Print { unPrint :: Int → Doc }
```

Tagless final + эффекты + символьное исполнение

- интересно для такого языка написать принтер
- реализовать MonadState для Print тривиально:

```
instance MonadState Int Print where
  get = Print $ const $ text "get"
  put x = Print $ const $ text "put" <+> PP.int x
```

- а вот с Monad у нас проблемы!

```
instance Monad Print where
  (>>=) :: Print a → (a → Print b) → Print b
  l >>= r = Print $ \c → unPrint l c <+> text ">>="
                <+> unPrint (r ???undefined???) c
```

- мы исполняем символьно --- нам неоткуда взять настоящее значение типа a

```
newtype Print a = Print { unPrint :: Int → Doc }
```

- все плохо, все потеряно?

Tagless final + эффекты + символьное исполнение, часть 2

- решение неочевидное, но очень простое

Tagless final + эффекты + символьное исполнение, часть 2

- решение неочевидное, но очень простое
- заметим, что зависимость по данным / «выход из монады» не только невозможен, но и *не нужен* для символьного исполнения

Tagless final + эффекты + символьное исполнение, часть 2

- решение неочевидное, но очень простое
- заметим, что зависимость по данным / «выход из монады» не только невозможен, но и *не нужен* для символьного исполнения
- наш язык не должен этого требовать (соотв. не подходит существующий `MonadState`)

Tagless final + эффекты + символьное исполнение, часть 2

- решение неочевидное, но очень простое
- заметим, что зависимость по данным / «выход из монады» не только невозможен, но и *не нужен* для символьного исполнения
- наш язык не должен этого требовать (соотв. не подходит существующий `MonadState`)
- но главное, надо абстрагироваться от способа композиции

Tagless final + эффекты + символьное исполнение, часть 2

- решение неочевидное, но очень простое
- заметим, что зависимость по данным / «выход из монады» не только невозможен, но и *не нужен* для символьного исполнения
- наш язык не должен этого требовать (соотв. не подходит существующий `MonadState`)
- но главное, надо абстрагироваться от способа композиции
    - самый общий и «безвыходный» случай --- это просто применение функций

```
(>>=)    :: repr a → (      a → repr b) → repr b
flip ($) :: repr a → (repr a → repr b) → repr b
```

Tagless final + эффекты + символьное исполнение, часть 2

- решение неочевидное, но очень простое
- заметим, что зависимость по данным / «выход из монады» не только невозможен, но и *не нужен* для символьного исполнения
- наш язык не должен этого требовать (соотв. не подходит существующий `MonadState`)
- но главное, надо абстрагироваться от способа композиции
  - самый общий и «безвыходный» случай --- это просто применение функций

```
(>>=)    :: repr a → (     a → repr b) → repr b
flip ($) :: repr a → (repr a → repr b) → repr b
```

- вот такой класс получается

```
class Combinators repr where
  obind :: repr a -> (repr a -> repr b) -> repr b
  oseq  :: repr a -> repr b -> repr b
```

Tagless final + эффекты + символьное исполнение, часть 3

- такое можно реализовать для всего

```haskell
class Combinators repr where
  obind :: repr a -> (repr a -> repr b) -> repr b
  oseq  :: repr a -> repr b -> repr b
```

Tagless final + эффекты + символьное исполнение, часть 3

- такое можно реализовать для всего

```
class Combinators repr where
  obind :: repr a -> (repr a -> repr b) -> repr b
  oseq  :: repr a -> repr b -> repr b
```

- для реального исполнения с монадами:

```
instance Monad m => Combinators m where
  l `obind` r = l >>= (r . return)
  oseq = (>>)
```

Tagless final + эффекты + символьное исполнение, часть 3

- такое можно реализовать для всего

```
class Combinators repr where
  obind :: repr a -> (repr a -> repr b) -> repr b
  oseq  :: repr a -> repr b -> repr b
```

- для реального исполнения с монадами:

```
instance Monad m => Combinators m where
  l `obind` r = l >>= (r . return)
  oseq = (>>)
```

- для печати:

```
instance {-# OVERLAPPING #-} Combinators Print where
  l `obind` r = Print $ \c → unPrint l c <+> text ">>="
                <+> parens (unPrint (lam r) (succ c))
  l `oseq` r = Print $ \c → unPrint l c <+> text ">>"
                <+> parens (unPrint r c)
```

Tagless final + эффекты + символьное исполнение, часть 4

- перепишем `MonadState` чтобы «оставлять все в языке»

```
class MyState s repr where
  mget :: repr s
  mput :: repr s → repr ()
```

Tagless final + эффекты + символьное исполнение, часть 4

- перепишем `MonadState` чтобы «оставлять все в языке»

```
class MyState s repr where
  mget :: repr s
  mput :: repr s → repr ()
```

- по-настоящему исполнить просто (монадическая композиция теперь спрятана тут)

```
instance MonadState s repr => MyState s repr where
  mget = get
  mput x = x >>= put
```

Tagless final + эффекты + символьное исполнение, часть 4

- перепишем MonadState чтобы «оставлять все в языке»

```
class MyState s repr where
  mget :: repr s
  mput :: repr s → repr ()
```

- по-настоящему исполнить просто (монадическая композиция теперь спрятана тут)

```
instance MonadState s repr => MyState s repr where
  mget = get
  mput x = x >>= put
```

- и символьно исполнить (напечатать) тоже просто

```
instance {-# OVERLAPPING #-} MyState s Print where
  mget = Print $ const $ text "mget"
  mput x = Print $ \c → text "mput" <+>
              parens (unPrint x c)
```

Tagless final + эффекты + символьное исполнение, часть 5

- {- не делайте `OVERLAPPING` в реальном коде, используйте `newtype`-обертки, чтобы не было инстансов для всех `m` -}

Tagless final + эффекты + символьное исполнение, часть 5

- {- не делайте OVERLAPPING в реальном коде, используйте newtype-обертки, чтобы не было инстансов для всех m -}
- оно работает!

```
prgSeq = mput (mget `add` int 1)
         `oseq` ((mget `add` int 2) `obind` mput)
main = putStrLn $ show (unPrint prgSeq 0)
-- mput (mget + 1) >> (mget + 2 >>= (\var1 → mput (var1)))
```

Tagless final + эффекты + символьное исполнение, часть 5

- `{-` не делайте `OVERLAPPING` в реальном коде, используйте `newtype`-обертки, чтобы не было инстансов для всех `m` `-}`
- оно работает!

```
prgSeq = mput (mget `add` int 1)
         `oseq` ((mget `add` int 2) `obind` mput)
main = putStrLn $ show (unPrint prgSeq 0)
-- mput (mget + 1) >> (mget + 2 >>= (\var1 → mput (var1)))
```

- забавная особенность такого подхода --- вместо `obind` можно просто передавать аргумент

Tagless final + эффекты + символьное исполнение, часть 5

- {- не делайте `OVERLAPPING` в реальном коде, используйте `newtype`-обертки, чтобы не было инстансов для всех `m` -}
- оно работает!

```
prgSeq = mput (mget `add` int 1)
         `oseq` ((mget `add` int 2) `obind` mput)
main = putStrLn $ show (unPrint prgSeq 0)
-- mput (mget + 1) >> (mget + 2 >>= (\var1 → mput (var1)))
```

- забавная особенность такого подхода --- вместо `obind` можно просто передавать аргумент
  - т.е. нам на самом деле нужен только комбинатор `oseq`

Tagless final + эффекты + символьное исполнение, часть 5

- {- не делайте OVERLAPPING в реальном коде, используйте newtype-обертки, чтобы не было инстансов для всех m -}
- оно работает!

```
prgSeq = mput (mget `add` int 1)
         `oseq` ((mget `add` int 2) `obind` mput)
main = putStrLn $ show (unPrint prgSeq 0)
-- mput (mget + 1) >> (mget + 2 >>= (\var1 → mput (var1)))
```

- забавная особенность такого подхода --- вместо obind можно просто передавать аргумент
    - т.е. нам на самом деле нужен только комбинатор oseq
    - но на практике в сложных случаях obind помогает с выводом типов

Tagless final + эффекты + символьное исполнение, часть 5

- {- не делайте `OVERLAPPING` в реальном коде, используйте `newtype`-обертки, чтобы не было инстансов для всех `m` -}
- оно работает!

```
prgSeq = mput (mget `add` int 1)
         `oseq` ((mget `add` int 2) `obind` mput)
main = putStrLn $ show (unPrint prgSeq 0)
-- mput (mget + 1) >> (mget + 2 >>= (\var1 → mput (var1)))
```

- забавная особенность такого подхода --- вместо `obind` можно просто передавать аргумент
  - т.е. нам на самом деле нужен только комбинатор `oseq`
  - но на практике в сложных случаях `obind` помогает с выводом типов
- проблема: код выглядит страшно, кто хочет писать `oseq` вместо переноса строки в do-блоке??

Перегрузка стандартного синтаксиса

- `RebindableSyntax` спешит на помощь!

Перегрузка стандартного синтаксиса

- `RebindableSyntax` спешит на помощь!
- с этим расширением очень много чего берется из текущего scope, а не из стандартной библиотеки

Перегрузка стандартного синтаксиса

- `RebindableSyntax` спешит на помощь!
- с этим расширением очень много чего берется из текущего scope, а не из стандартной библиотеки
- т.е. do-нотация будет использовать локальные >> и >>=:

```
prgSeq = do mput (mget `add` int 1)
            (mget `add` int 2) >>= mput
         where (>>) = oseq
               (>>=) = obind
```

Перегрузка стандартного синтаксиса

- `RebindableSyntax` спешит на помощь!
- с этим расширением очень много чего берется из текущего scope, а не из стандартной библиотеки
- т.е. do-нотация будет использовать локальные >> и >>=:

```
prgSeq = do mput (mget `add` int 1)
            (mget `add` int 2) >>= mput
       where (>>) = oseq
             (>>=) = obind
```

- у нас есть трюк, позволяющий легко импортировать сразу много таких штук. О нем не будем :)

Еще больше перегрузки

Стоит знать о всяких других возможностях перегрузки:

- в `RebindableSyntax` входит в том числе `if-then-else`

Еще больше перегрузки

Стоит знать о всяких других возможностях перегрузки:

- в `RebindableSyntax` входит в том числе `if-then-else`
- а вот `case` перегружать нельзя

Еще больше перегрузки

Стоит знать о всяких других возможностях перегрузки:

- в `RebindableSyntax` входит в том числе `if-then-else`
- а вот `case` перегружать нельзя
    - но `ViewPatterns` может помочь?

Еще больше перегрузки

Стоит знать о всяких других возможностях перегрузки:

- в `RebindableSyntax` входит в том числе `if-then-else`
- а вот `case` перегружать нельзя
  - но `ViewPatterns` может помочь?
- числовые константы и операции берутся из класса `Num`

Еще больше перегрузки

Стоит знать о всяких других возможностях перегрузки:

- в `RebindableSyntax` входит в том числе `if-then-else`
- а вот `case` перегружать нельзя
  - но `ViewPatterns` может помочь?
- числовые константы и операции берутся из класса `Num`
- с расширением `OverloadedStrings` строковые константы создаются через класс `IsString`

Еще больше перегрузки

Стоит знать о всяких других возможностях перегрузки:

- в `RebindableSyntax` входит в том числе `if-then-else`
- а вот `case` перегружать нельзя
    - но `ViewPatterns` может помочь?
- числовые константы и операции берутся из класса `Num`
- с расширением `OverloadedStrings` строковые константы создаются через класс `IsString`
- аналогично для списков `OverloadedLists` / `IsList`

Еще больше перегрузки

Стоит знать о всяких других возможностях перегрузки:

- в `RebindableSyntax` входит в том числе `if-then-else`
- а вот `case` перегружать нельзя
  - но `ViewPatterns` может помочь?
- числовые константы и операции берутся из класса `Num`
- с расширением `OverloadedStrings` строковые константы создаются через класс `IsString`
- аналогично для списков `OverloadedLists` / `IsList`
- `OverloadedLabels`

Как декомпозировать языки

- крайности: один большой класс, по классу на выражение

Как декомпозировать языки

- крайности: один большой класс, по классу на выражение
- адекватный вариант: по аспектам

Как декомпозировать языки

- крайности: один большой класс, по классу на выражение
- адекватный вариант: по аспектам
- позволяет вводить ограничения вида «`break` может быть только внутри `loop`»

```
class RBreak repr where
  brk :: repr ()

class RBreak subreprL
   => RLoop subreprL repr
    | repr → subreprL where
  loop :: subreprL () → repr ()
```

Немного про реализацию интерпретаторов-исполнителей

Monad Transformers

- самая популярная/старая/простая система эффектов

Monad Transformers

- самая популярная/старая/простая система эффектов
- позволяет «наслаивать» эффекты: `ExceptT MyErr (StateT MySt IO) a`

Monad Transformers

- самая популярная/старая/простая система эффектов
- позволяет «наслаивать» эффекты: `ExceptT MyErr (StateT MySt IO) a`
- операция `lift` «заворачивает» вычисление во внутренней монаде

```
class MonadTrans (t :: (* → *) → * → *) where
  lift :: Monad m => m a → t m a
```

Monad Transformers

- самая популярная/старая/простая система эффектов
- позволяет «наслаивать» эффекты: `ExceptT MyErr (StateT MySt IO) a`
- операция `lift` «заворачивает» вычисление во внутренней монаде

```
class MonadTrans (t :: (* → *) → * → *) where
  lift :: Monad m => m a → t m a
```

- сами трансформеры --- просто `newtype`-обертки:

```
newtype ReaderT r m a = ReaderT{runReaderT :: r → m a }
newtype StateT  s m a = StateT {runStateT  :: s → m (a,s)}
```

Monad Transformers

- самая популярная/старая/простая система эффектов
- позволяет «наслаивать» эффекты: `ExceptT MyErr (StateT MySt IO) a`
- операция `lift` «заворачивает» вычисление во внутренней монаде

```
class MonadTrans (t :: (* → *) → * → *) where
  lift :: Monad m => m a → t m a
```

- сами трансформеры --- просто `newtype`-обертки:

```
newtype ReaderT r m a = ReaderT{runReaderT :: r → m a }
newtype StateT  s m a = StateT {runStateT  :: s → m (a,s)}
```

- они реализуют классы `MonadReader`, `MonadState` etc. из `mtl`

newtype-обертки

- можно объявлять свои `newtype`-обертки в т.ч. используя трансформеры внутри:

```
newtype LoopT (repr :: * → *) (a :: *) =
  LoopT { unLoopT :: ExceptT () repr a }
  deriving (Functor, Applicative, Monad)
```

newtype-обертки

- можно объявлять свои `newtype`-обертки в т.ч. используя трансформеры внутри:

```
newtype LoopT (repr :: * → *) (a :: *) =
  LoopT { unLoopT :: ExceptT () repr a }
  deriving (Functor, Applicative, Monad)
```

- GHC может выводить много чего автоматически (GeneralizedNewtypeDeriving)

newtype-обертки

- можно объявлять свои `newtype`-обертки в т.ч. используя трансформеры внутри:

```
newtype LoopT (repr :: * → *) (a :: *) =
  LoopT { unLoopT :: ExceptT () repr a }
  deriving (Functor, Applicative, Monad)
```

- GHC может выводить много чего автоматически (GeneralizedNewtypeDeriving)
  - иногда тупит с `MonadTrans`, но его руками легко написать:

```
instance MonadTrans LoopT where
  lift = LoopT . lift
```

newtype-обертки

- можно объявлять свои `newtype`-обертки в т.ч. используя трансформеры внутри:

```
newtype LoopT (repr :: * → *) (a :: *) =
  LoopT { unLoopT :: ExceptT () repr a }
  deriving (Functor, Applicative, Monad)
```

- GHC может выводить много чего автоматически (GeneralizedNewtypeDeriving)
  - иногда тупит с `MonadTrans`, но его руками легко написать:

```
instance MonadTrans LoopT where
  lift = LoopT . lift
```

- GHC может выводить инстансы для классов с реализациями по умолчанию для всего (DeriveAnyClass)

newtype-обертки

- можно объявлять свои newtype-обертки в т.ч. используя трансформеры внутри:

```
newtype LoopT (repr :: * → *) (a :: *) =
  LoopT { unLoopT :: ExceptT () repr a }
  deriving (Functor, Applicative, Monad)
```

- GHC может выводить много чего автоматически (GeneralizedNewtypeDeriving)
  - иногда тупит с MonadTrans, но его руками легко написать:

```
instance MonadTrans LoopT where
  lift = LoopT . lift
```

- GHC может выводить инстансы для классов с реализациями по умолчанию для всего (DeriveAnyClass)
- GHC может автоматизировать copy-paste (DerivingVia)

DerivingVia

- объявляем newtype-«стратегию»:

```
newtype RLoopM (f :: * → *) (a :: *) = RLoopM (f a)
  deriving (Functor, Applicative, Monad)

instance Monad repr
      => RLoop (LoopT repr) (RLoopM repr) where
  loop x = coerce @(repr ()) $ void $
             runExceptT $ forever $ unLoopT x
```

DerivingVia

- объявляем `newtype`-«стратегию»:

```
newtype RLoopM (f :: * → *) (a :: *) = RLoopM (f a)
  deriving (Functor, Applicative, Monad)

instance Monad repr
     => RLoop (LoopT repr) (RLoopM repr) where
  loop x = coerce @(repr ()) $ void $
                runExceptT $ forever $ unLoopT x
```

- говорим компилятору копировать реализации оттуда:

```
newtype RejectT (repr :: * → *) (a :: *) =
  RejectT { unRejectT :: ExceptT ExReason repr a }
  deriving ( RLoop (LoopT (RejectT repr)) )
       via RLoopM (RejectT repr)
```

Спасибо