CS314. Функциональное программирование

Лекции 14–15. Преобразователи монад

Направление «Фундаментальная информатика и информационные технологии»
Институт математики, механики и компьютерных наук имени И. И. Воровича
Южный федеральный университет

10 ноября 2017 г.

Содержание

Содержание

Запрос пароля

Постановка задачи

При регистрации в некоторой системе пользователь должен указать пароль, который он будет использовать в дальнейшем для аутентификации.
Необходимо организовать чтение предлагаемого пользователем пароля с клавиатуры, совместив его с проверкой, является ли он достаточно стойким.

Проверка стойкости

```
isValid :: String -> Bool
isValid s =
    length s >= 8
    && any isAlpha s
    && any isNumber s
    && any isPunctuation s
```

Используемые монады

- `IO` — ввод-вывод;
- `Maybe` — `Just <пароль>` либо `Nothing` (возможная неудача — пароль не прошёл проверку на стойкость).

Запрос пароля: вариант решения

Чтение пароля

```
getPassword = do
    putStrLn "Введите новый пароль:"
    s <- getLine
    return (if isValid s then Just s else Nothing)

askPassword = do
    mpasswd <- getPassword
    if isJust mpasswd
        then do
            putStrLn "Сохранение..."
            -- необходимые действия
        else
            askPassword
```

- В какой монаде исполняется этот код?
- Используются ли здесь возможности монады Maybe?

Ответы: IO, нет.

Комбинирование монад

- Хочется иметь средство для комбинирования монад: то есть уметь создавать монаду, объединяющую возможности двух или более имеющихся монад.
- Оказывается, не любые две монады можно скомбинировать.
- Не существует общего способа скомбинировать две произвольные монады.
- Можно написать код, добавляющий возможности некоторой конкретной монады к любой другой. Например: монада IO с возможной неудачей в вычислениях.
- Основные понятия: преобразователи монад (monad transformers), стек монад, подъём по стеку.

Решение с построением стека монад

```
getPassword :: MaybeT IO String
getPassword = do
  lift $ putStrLn
       "Введите новый пароль:"
  s <- lift getLine
  guard (isValid s)
  return s

askPassword :: MaybeT IO ()
askPassword = do
  value <- msum $
           repeat getPassword
  lift $ putStrLn "Сохранение..."

main = runMaybeT askPassword
```

Компоненты решения

- MaybeT — преобразователь монад.
- MaybeT IO String — стек монад (тоже монада).
- Maybe — базовая монада, IO — внутренняя монада.
- lift — «поднятие» функции до внутренней монады.
- guard, msum — возможности экземпляра MonadPlus для Maybe.
- runMaybeT — запуск новой монады.

Импортируемые модули и типы

```
import Control.Monad
import Control.Monad.Trans
import Control.Monad.Trans.Maybe
```

```
getPassword :: MaybeT IO String
askPassword :: MaybeT IO ()
```

```
lift :: (Monad m, MonadTrans t) => m a -> t m a
msum :: (Foldable t, MonadPlus m) => t (m a) -> m a
runMaybeT :: MaybeT m a -> m (Maybe a)
```

Содержание

Преобразователи стандартных монад

- MaybeT, ReaderT, WriterT, StateT, ...

Пример: версия 1

```
m :: MaybeT (ReaderT Integer IO) String
m = do
  n <- lift ask
  guard (n > 0)
  lift $ lift $ print n
  return "OK"

main = runReaderT (runMaybeT m) 2
```

- Базовая монада — Maybe.
- Внутренняя монада — IO.
- Стек содержит три монады.
- Поднятие (lift).

Автоматическое поднятие

Пример: версия 2 с автоматическим поднятием

```
m :: MaybeT (ReaderT Integer IO) String
m = do
  n <- ask
  guard (n > 0)
  liftIO $ print n
  return "OK"

main = runReaderT (runMaybeT m) 2
```

- Функция `lift` вызывается автоматически (в зависимости от структуры стека монад).
- Функция `liftIO` всегда поднимает до внутренней монады `IO`.

Когда необходимо явное поднятие?

```
m :: StateT Integer (State String) ()
m = do
  n <- get
  s <- lift $ get
  return ()
```

```
m2 :: ReaderT Bool (StateT Integer (State String)) ()
m2 = do
  b <- ask
  n <- lift $ get
  s <- lift $ lift $ get
  return ()
```

- Необходимы явные обёртки.
- Слишком жёсткий расчёт на структуру стека монад.

Содержание

Монада Identity

```haskell
newtype Identity a = Identity { runIdentity :: a }

instance Functor Identity where
    fmap :: (a->b) -> Identity a -> Identity b
    fmap     = coerce

instance Applicative Identity where
    pure     = Identity
    (<*>) :: Identity (a->b) -> Identity a -> Identity b
    (<*>)    = coerce

instance Monad Identity where
    return   = Identity
    m >>= k  = k (runIdentity m)
```

```haskell
coerce :: Coercible a b => a -> b
```

Базовые классы-преобразователи

Класс MonadTrans

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

Класс MonadIO

```
class (Monad m) => MonadIO m where
    liftIO :: IO a -> m a

instance MonadIO IO where
    liftIO = id
```

Преобразователь MaybeT

```haskell
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

instance MonadTrans MaybeT where
    lift :: (Monad m) => m a -> MaybeT m a
    lift = MaybeT . liftM Just

instance (MonadIO m) => MonadIO (MaybeT m) where
    liftIO = lift . liftIO
```

Преобразователь MaybeT создаёт монаду

```
instance (Monad m) => Monad (MaybeT m) where
    return = lift . return
    x >>= f = MaybeT $ do
        v <- runMaybeT x
        case v of
            Nothing -> return Nothing
            Just y  -> runMaybeT (f y)
```

- В какой монаде работает код в блоке do?

Монада Writer и преобразователь WriterT

```haskell
type Writer w = WriterT w Identity
```

```haskell
newtype WriterT w m a = WriterT { runWriterT :: m (a, w) }
```

```haskell
instance (Monoid w, Monad m) => Monad (WriterT w m) where
    return a = writer (a, mempty)
    m >>= k  = WriterT $ do
        (a, w)  <- runWriterT m
        (b, w') <- runWriterT (k a)
        return (b, w `mappend` w')
```

Монада Writer и преобразователь WriterT (2)

```haskell
instance (Monoid w) => MonadTrans (WriterT w) where
    lift m = WriterT $ do
        a <- m
        return (a, mempty)

instance (Monoid w, MonadIO m) => MonadIO (WriterT w m) where
    liftIO = lift . liftIO
```

- Как сделать так, чтобы другие монады могли воспользоваться возможностями монады Writer?

Класс MonadWriter и его использование

```
class (Monoid w, Monad m) => MonadWriter w m | m -> w where
    writer :: (a,w) -> m a
    tell   :: w -> m ()
    listen :: m a -> m (a, w)
    pass   :: m (a, w -> w) -> m a
```

```
instance MonadWriter w m => MonadWriter w (MaybeT m) where
    writer = lift . writer
    tell   = ...
    listen = ...
    pass   = ...

instance MonadWriter w m => MonadWriter w (ReaderT r m) where
    ...
instance MonadWriter w m => MonadWriter w (StateT s m) where
    ...
```

Вернёмся к примеру

Пример: версия 2 с автоматическим поднятием

```
m :: MaybeT (ReaderT Integer IO) String
m = do
  n <- ask
  guard (n > 0)
  liftIO $ print n
  return "OK"
```

- Есть класс `MonadReader` с функцией `ask`.
- Есть экземпляр:

```
instance MonadReader r m => MonadReader r (MaybeT m) where
    ask    = lift ask
```

- Есть `MonadIO` для `MaybeT` и `ReaderT`, поэтому `liftIO` превращается в `lift.liftIO` и далее в lift.(lift.liftIO) = lift.lift.id.

Итоги

- Монада Identity.
- Монада есть преобразователь над Identity.
- Преобразователь создаёт обёртку над произвольной монадой, которая сама оказывается монадой, при этом bind базовой (новой) монады вызывает bind внутренней.
- Возможности монад из строящегося стека (классы MonadReader, MonadWriter и пр.) используются либо вручную с помощью `lift` из `MonadTrans`, либо автоматически с помощью экземпляров `MonadIO` и попарных реализаций вида

```
instance MonadWriter w m => MonadWriter w (ReaderT r m) where

instance (Monoid w, MonadReader r m) =>
                              MonadReader r (WriterT w m) where
```

Содержание

Подсчёт количества файлов в дереве каталогов

Задача

Даны имя каталога и глубина поиска. Посчитать количество файлов в каждом подкаталоге данного каталога, опускаясь не более чем на заданную глубину в дереве каталогов.

Компоненты решения

- Конфигурация: максимальная глубина.
- Состояние: текущая глубина, имя каталога.
- Результат: журнал со списком посещённых каталогов и количеством файлов в них.
- Используемые монады: IO, Writer, Reader, State.

Решение: используемые расширения и модули

```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
import System.Environment
import System.Directory
import System.FilePath
import Control.Monad.Writer
import Control.Monad.Reader
import Control.Monad.State
import Control.Applicative
```

Решение: конфигурация, состояние и журнал

```haskell
data AppConfig = AppConfig {
      cfgMaxDepth :: Int
    } deriving (Show)

data AppState = AppState {
      stCurDepth :: Int,
      stCurPath :: FilePath
    } deriving (Show)

type AppLog = [(FilePath, Int)]
```

Решение: стек монад

```
newtype MyApp a = MyA {
      runA :: ReaderT AppConfig (StateT AppState
                                 (WriterT AppLog IO)) a
    } deriving (Functor, Applicative, Monad,
                MonadIO,
                MonadReader AppConfig,
                MonadWriter [(FilePath, Int)],
                MonadState AppState)

runMyApp :: MyApp a -> Int -> FilePath -> IO (a, AppLog)
runMyApp app maxDepth path =
    let config = AppConfig maxDepth
        state = AppState 0 path
    in runWriterT (
           evalStateT (
              runReaderT (runA app) config) state)
```

Решение: вспомогательная функция

```haskell
listDirectory :: FilePath -> IO [String]
listDirectory d = filter notDots <$>  getDirectoryContents d
    where notDots p = p /= "." && p /= ".."
```

Решение: вычисление в монаде MyApp

```haskell
constrainedCount :: MyApp ()
constrainedCount = do
  maxDepth <- cfgMaxDepth <$> ask
  st <- get
  let curDepth = stCurDepth st
  when (curDepth < maxDepth) $ do
    let path = stCurPath st
    contents <- liftIO $ listDirectory path
    tell [(path, length contents)]
    forM_ contents $ \name -> do
      let newPath = path </> name
      isDir <- liftIO $ doesDirectoryExist newPath
      when isDir $ do
        put $ st {stCurDepth = curDepth + 1,
                  stCurPath = newPath}
        constrainedCount
```

Решение: запуск вычисления в функции main

```
main = do
  [md, p] <- getArgs
  (_, xs) <- runMyApp constrainedCount (read md) p
  print xs
```

Пример запуска

```
$ ./count 2 "."
[(".",4),("./A",1),("./B",0)]
$ ./count 3 "."
[(".",4),("./A",2),("./A/B2",0),("./A/A2",2),("./B",0)]
```

Результаты

Что в итоге?

Разработана схема приложения со следующими возможностями:

- чтение конфигурационной информации;
- запись журнала;
- использование изменяемого состояния;
- работа с вводом-выводом.

Гибкость полученного решения

- Возможно расширение конфигурационной информации.
- Возможно расширение состояния.
- Возможно изменение формата журнала.

Содержание

Постановка задачи (Constraints Satisfaction Problem)

Имеется логическая задача с переменными и ограничениями в форме предикатов. Необходимо установить возможные значения переменных, удовлетворяющие всем ограничениям.

Пример задачи

В племени калотанцев есть давний обычай: их мужчины всегда говорят только правду, тогда как женщины никогда не говорят правду или неправду два раза подряд. Антрополог Джонс начал изучать племя калотанцев, хотя с их языком пока не знаком. Однажды он встречает разнополую пару калотанцев с ребёнком Киби. Джонс спрашивает Киби: «А ты мальчик?», на что ребёнок отвечает на калотанском, которого Джонс не понимает. Затем Джонс обращается к родителям, знающим английский, за разъяснениями. Один из них говорит: «Киби сказал, что он мальчик». Другой же сообщает: «Киби девочка, она соврала». Каков пол Киби и каждого из родителей?

Идеи решения

- Переменные: родители и ребёнок, их значения — пол (мужской/женский).
- Формулировка необходимых предикатов (как общих, так и для конкретной задачи) => множество ограничений.
- Перебор множества решений (подстановок значений переменных) и проверка каждого на предмет удовлетворения множества ограничений.
- Списковая монада для реализации недетерминированных вычислений и фильтрации.
- Монада State для работы с переменными.

Переменные и предикаты

```
type Var = String
type Value = String
data Predicate =
             Is Var Value
           | Equal Var Var
           | And Predicate Predicate
           | Or Predicate Predicate
           | Not Predicate
  deriving (Eq, Show)

type Variables = [(Var, Value)]
```

Составные предикаты

```
isNot :: Var -> Value -> Predicate
isNot var value = Not (Is var value)

implies :: Predicate -> Predicate -> Predicate
implies a b = Not a `Or` b

orElse :: Predicate -> Predicate -> Predicate
orElse a b = (a `And` (Not b)) `Or` ((Not a) `And` b)
```

Проверка значения предиката

```
check :: Variables -> Predicate -> Maybe Bool
check vars (Is var val) = liftM (==val) (lookup var vars)
check vars (Equal v1 v2) = liftM2 (==) (lookup v1 vars)
                                       (lookup v2 vars)
check vars (And p1 p2) = liftM2 (&&) (check vars p1)
                                     (check vars p2)
check vars (Or  p1 p2) = liftM2 (||) (check vars p1)
                                     (check vars p2)
check vars (Not p) = liftM not (check vars p)
```

Состояние задачи

```
data ProblemState = PS {vars :: Variables,
                        constraints :: [Predicate]}

findVar :: Var -> ProblemState -> Maybe Value
findVar v (PS vs _) = lookup v vs

updateVar :: Var -> Value -> ProblemState -> ProblemState
updateVar v x (PS vs cs)
            = PS ((v, x) : filter ((v/=).fst) vs) cs
```

Состояние + список = недетерминированное состояние

```haskell
type NDS a = StateT ProblemState [] a

getVar :: Var -> NDS (Maybe Value)
getVar v = findVar v <$> get

setVar :: Var -> Value -> NDS ()
setVar v x = modify (updateVar v x)
```

```haskell
get :: m s
gets :: MonadState s m => (s -> a) -> m a
modify :: MonadState s m => (s -> s) -> m ()
```

Проверка удовлетворения ограничений

```
isConsistent :: Bool -> NDS Bool
isConsistent partial = do
  cs <- gets constraints
  vs <- gets vars
  return $ all (maybe partial id . check vs) cs

getFinalVars :: NDS Variables
getFinalVars = do
  c <- isConsistent False
  guard c
  gets vars
```

- Решение частично согласовано (partial), если не все переменные определены.

```
maybe :: b -> (a -> b) -> Maybe a -> b
```

Поиск решения (перебор значений и результаты)

```
tryAllValues :: Var -> [Value] -> NDS ()
tryAllValues var values = do
  msum $ map (setVar var) values
  c <- isConsistent True
  guard c
```

```
getAllSolutions :: ProblemState -> NDS a -> [a]
getAllSolutions ps c = evalStateT c ps

getSolution :: ProblemState -> NDS a -> Maybe a
getSolution ps c = listToMaybe (evalStateT c ps)
```

Формулировка задачи на естественном языке

В племени калотанцев есть давний обычай: их мужчины всегда говорят только правду, тогда как женщины никогда не говорят правду или неправду два раза подряд. Антрополог Джонс начал изучать племя калотанцев, хотя с их языком пока не знаком. Однажды он встречает разнополую пару калотанцев с ребёнком Киби. Джонс спрашивает Киби: «А ты мальчик?», на что ребёнок отвечает на калотанском, которого Джонс не понимает. Затем Джонс обращается к родителям, знающим английский, за разъяснениями. Один из них говорит: «Киби сказал, что он мальчик». Другой же сообщает: «Киби девочка, она соврала». Каков пол Киби и каждого из родителей?

Специфичные для задачи предикаты

```
said :: Var -> Predicate -> Predicate
said v p = (v `Is` "male") `implies` p

saidBoth :: Var -> Predicate -> Predicate -> Predicate
saidBoth v p1 p2 =
     And ((v `Is` "male") `implies` (p1 `And` p2))
         ((v `Is` "female") `implies` (p1 `orElse` p2))

lied :: Var -> Predicate -> Predicate
lied v p = ((v `said` p) `And` (Not p))
                        `orElse` ((v `said` (Not p)) `And` p)
```

Формулировка задачи на Haskell

```
main = do
  let variables = []
      values = ["male", "female"]
      constraints = [
        Not (Equal "parent1" "parent2"),
        "parent1" `said` ("child" `said`
                              ("child" `Is` "male")),
        saidBoth "parent2" ("child" `Is` "female")
                           ("child" `lied`
                              ("child" `Is` "male"))]
       problem = PS variables constraints
  print $ getSolution problem $ do
     tryAllValues "parent1" values
     tryAllValues "parent2" values
     tryAllValues "child" values
     getFinalVars
```

Источники примеров

1. Запрос пароля: Haskell/Monad transformers на wikibooks (`http://en.wikibooks.org/wiki/Haskell/Monad_transformers`)
2. Приложение с конфигурацией, состоянием и журналом: глава 18 из Real World Haskell (`http://book.realworldhaskell.org/read/monad-transformers.html`)
3. Логическая задача: пункт 22.3 StateT with List из All About Monads (`https://www.haskell.org/haskellwiki/All_About_Monads#StateT_with_List`)

Решение задачи о калотанцах

```
ghci> :main
Just [("child","female"),
      ("parent2","male"),
      ("parent1","female")]
```