Free монады

Косарев Дмитрий a.k.a. Kakadu

матмех СПбГУ

6 декабря 2018 г.

Содержание

Outline

Очередной мини язык

- `output b` – печатает "b"
- `bell` – звенеть как `echo -e "\a"`
- `done` – конец исполнения

Очередной мини язык

- `output b` – печатает "b"
- `bell` – звенеть как `echo -e "\a"`
- `done` – конец исполнения

Заведем тип для программы, предварительно вот так:

```
data Toy b next = Output b next
                | Bell next
                | Done
```

`Done` всегда последняя и будет означать конец исполнения

Теперь мы умеем конструировать программы

```
data Toy b next = Output b next
                | Bell next
                | Done

-- output 'A'
-- done
Output 'A' Done
           :: Toy Char (Toy a next)

-- bell
-- output 'A'
-- done
Bell (Output 'A' Done)
           :: Toy a (Toy Char (Toy b next)))
```

Теперь мы умеем конструировать программы

```
data Toy b next = Output b next
                | Bell next
                | Done

-- output 'A'
-- done
Output 'A' Done
           :: Toy Char (Toy a next)

-- bell
-- output 'A'
-- done
Bell (Output 'A' Done)
           :: Toy a (Toy Char (Toy b next)))
```

Проблема: для разных программ разный тип.

Применим чит

```
data Cheat f = Cheat (f (Cheat f))
```

Теперь у нас программы будут типизироваться всегда одинаково

```
Cheat (Output 'A' (Cheat Done))
  :: Cheat (Toy Char)

Cheat (Bell (Cheat (Output 'A' (Cheat Done))))
  :: Cheat (Toy Char)
```

Вот только это немножко велосипед, потому что такое в Haskell уже есть.

Фикс из модуля Data.Fix

```
data Fix f = Fix (f (Fix f))
```

```
Fix (Output 'A' (Fix Done))
  :: Fix (Toy Char)
```

```
Fix (Bell (Fix (Output 'A' (Fix Done))))
  :: Fix (Toy Char)
```

Уже лучше, но есть другая проблема. Программы на мини-языке надо писать *до конца*.

Поведение при окончании программы

Будем в "кидать исключение", когда исполнение должно закончиться

```
data FixE f e = Fix (f (FixE f e)) | Throw e
```

А те, кто исполняют программу, будут исключения "ловить"

```
catch :: (Functor f) =>
  FixE f e1 -> (e1 -> FixE f e2) -> FixE f e2
catch (Fix x) f = Fix (fmap (flip catch f) x)
catch (Throw e) f = f e
```

Нам будет нужен функтор.

```
instance Functor (Toy b) where
    fmap f (Output x next) = Output x (f next)
    fmap f (Bell     next) = Bell     (f next)
    fmap f  Done           = Done
```

```
data IncompleteException = IncompleteExc

-- output 'A'
-- throw IncompleteExc
subroutine = Fix (Output 'A' (Throw IncompleteExc))
    :: FixE (Toy Char) IncompleteException

-- try { subroutine }
-- catch (IncompleteExc) {
--     bell
--     done
-- }
program :: FixE (Toy Char) e
program = subroutine `catch`
            (\_ -> Fix (Bell (Fix Done))
    :: FixE (Toy Char) e
```

Outline

Проблемка

```
data FixE f e = Fix (f (FixE f e)) | Throw e
```

Мы зарелизили библиотеку, но пользователи используют Throw, чтобы передавать нормальные результаты программ, а не моделировать исключительные ситуации.

Проблемка

```
data FixE f e = Fix (f (FixE f e)) | Throw e
```

Мы зарелизили библиотеку, но пользователи используют Throw, чтобы передавать нормальные результаты программ, а не моделировать исключительные ситуации.

Наверное потому, что мы навелосипедили опять!

```
data Free f r = Free (f (Free f r)) | Pure r
```

Ну, вы знаете, что я сейчас скажу

```
data Free f r = Free (f (Free f r)) | Pure r
```

Ну, вы знаете, что я сейчас скажу

```
data Free f r = Free (f (Free f r)) | Pure r

instance (Functor f) => Monad (Free f) where
    return = Pure
    (Free x) >>= f = Free (fmap (>>= f) x)
    (Pure r) >>= f = f r
```

- return $\sim$ Throw
- (>>=) $\sim$ catch

И так как это монада, то у нас будет do-нотация за бесплатно (англ. for free).

Немного сахара для конструирования программ

```
output x = Free (Output x (Pure ()))

bell :: Free (Toy a) ()
bell = Free (Bell (Pure ()))

done :: Free (Toy a) r
done = Free Done


liftF :: (Functor f) => f r -> Free f r
liftF command = Free (fmap Pure command)

output x = liftF (Output x ())
bell     = liftF (Bell     ())
done     = liftF  Done
```

Всё просто работает

```
subroutine :: Free (Toy Char) ()
subroutine = output 'A'

program :: Free (Toy Char) r
program = do
    subroutine
    bell
    done
```

Раньше монады использовались только чтобы имитировать эффекты, а теперь – чтобы *конструировать* данные.

Покажем, что это действительно данные, написав интерпретатор, который распечатывает

```
showProgram :: (Show a, Show r) =>
                    Free (Toy a) r -> String

showProgram (Free (Bell x)) =
                              "bell\n" ++ showProgram x
showProgram (Free Done) = "done\n"
showProgram (Pure r)    = "return " ++ show r ++ "\n"
showProgram (Free (Output a x)) =
       "output " ++ show a ++ "\n" ++ showProgram x
```

```
>>> putStr (showProgram program)
output 'A'
bell
done
```

А что там с законами? (1/3)

```
pretty :: (Show a, Show r) => Free (Toy a) r -> IO ()
pretty = putStr . showProgram
```

✔
```
>>> pretty (output 'A')
output 'A'
return ()
```

А что там с законами? (2/3)

? `>>> pretty (return 'A' >>= output)`

А что там с законами? (2/3)

```
?    >>> pretty (return 'A' >>= output)

     output 'A'
     return ()

?    >>> pretty (output 'A' >>= return)
```

А что там с законами? (2/3)

? `>>> pretty (return 'A' >>= output)`

```
output 'A'
return ()
```

? `>>> pretty (output 'A' >>= return)`

```
output 'A'
return ()
```

А что там с законами? (3/3)

```
? >>> pretty ((output 'A' >> done) >> output 'C')
```

А что там с законами? (3/3)

```
? >>> pretty ((output 'A' >> done) >> output 'C')

  output 'A'
  return ()

? >>> pretty (output 'A' >> (done >> output 'C'))
```

А что там с законами? (3/3)

```
? >>> pretty ((output 'A' >> done) >> output 'C')

  output 'A'
  return ()

? >>> pretty (output 'A' >> (done >> output 'C'))

  output 'A'
  return ()
```

Можно и нормальный интерпретатор

Будем использовать функцию `ringbell` из сторонней библиотеки, которая на самом деле будет "звенеть".

```
ringBell :: IO ()

interpret :: (Show b) => Free (Toy b) r -> IO ()

interpret (Free (Output b x)) = print b  >> interpret
interpret (Free (Bell     x)) = ringBell >> interpret
interpret (Free  Done       ) = return ()

interpret (Pure r) =
  throwIO (userError "Unexpected termination")
```

Дляя Free монады всёравно как вы её используете.

Outline

Если хотим concurrency для монады IO, то можно вызывать `forkIO` из модуля Control.Concurrent.

А если хотим это сделать для других монад: State или Cont?
Первое желание: список монадических "действий".

```
type Thread m = [m ()]
```

Но

- Тут нет информации о порядке вызывов
- И непонятно, где тут результат работы.

Один конструктор, чтобы интерпретатор знал что за чем делать.
Второй для результата

```
data Thread m r = Atomic (m (Thread m r)) | Return r
```

Один конструктор, чтобы интерпретатор знал что за чем делать.
Второй для результата

```
data Thread m r = Atomic (m (Thread m r)) | Return r
```

Вычисление одного шага будет выглядеть так:

```
atomic :: (Monad m) => m a -> Thread m a
atomic m = Atomic (liftM Return m)
```

где

```
liftM :: Monad m => (a1 -> r) -> m a1 -> m r
```

Разумеется

```
instance (Monad m) => Monad (Thread m) where
    return              = Return
    (Atomic m) >>= f = Atomic (liftM (>>= f) m)
    (Return r) >>= f = f r
```

Сконструируем пару кооперативных задач

```
thread1 :: Thread IO ()
thread1 = do
    atomic (print 1)
    atomic (print 2)

thread2 :: Thread IO ()
thread2 = do
    str <- atomic getLine
    atomic (putStrLn str)
```

Теперь надо научиться их исполнять.

```
interleave :: (Monad m) =>
  Thread m r -> Thread m r -> Thread m r

interleave (Atomic m1) (Atomic m2) = do
    next1 <- atomic m1
    next2 <- atomic m2
    interleave next1 next2

interleave t1 (Return _) = t1
interleave (Return _) t2 = t2
```

```
runThread :: (Monad m) => Thread m r -> m r
runThread (Atomic m) = m >>= runThread
runThread (Return r) = return r
```

```
>>> runThread (interleave thread1 thread2)
1
[[Input: "Hello, world!"]]
2
Hello, world!
```

Здесь в квадратных скобках – то, что вводилось человеком с клавиатуры.

Outline

Ну, вы заметили, что `Thread` – это `Free`, а `atomic` – это `liftF`.

А пример с тредами говорит, что `Free` жутко напоминает `List`.

```
data Free f r = Free (f (Free f r)) | Pure r
data List a   = Cons  a (List a  )  | Nil
```

В некотором смысле, `Free` – это список(`List`) функторов.

Лирическое отступление о том, когда может понадобиться free list и вообще free object.

Ну, и можно выражать одно через другое

```
type List' a = Free ((,) a) ()
```

$$
\begin{aligned}
\texttt{List' a} &= \texttt{Free ((,) a) ()} \\
&= \texttt{Free (a, List' a) | Pure ()} \\
&\sim \texttt{Free a (List' a) | Pure ()}
\end{aligned}
$$

У нас был instance монады для Free.

```
data Free f r = Free (f (Free f r)) | Pure r

instance (Functor f) => Monad (Free f) where
    return          = Pure
    (Free x) >>= f = Free (fmap (>>= f) x)
    (Pure r) >>= f = f r
```

Следовательно, у нас есть инстанс для

```
type List' a = Free ((,) a) ()
```

И ещё мы знаем, что `[]` это тоже монада.

? Одинаковые ли ведут себя инстансы для двух типов:

- `type List' a = Free ((,) a) ()` и
- `[]`

Посмотрим на Free как на список

```
singleton x = Cons x Nil    -- i.e. x:[], or [x]
liftF x = Free (fmap Pure x)
```

Посмотрим на Free как на список

```
singleton x = Cons x Nil    -- i.e. x:[], or [x]
liftF x = Free (fmap Pure x)

merge (x1:xs1) (x2:xs2) = x1:x2:merge xs1 xs2
merge xs1 [] = xs1
merge [] xs2 = xs2

-- [x1] ++ [x2] ++ interleave xs1 xs2
interleave (Atomic m1) (Atomic m2) = do
    next1 <- liftF m1
    next2 <- liftF m2
    interleave next1 next2
interleave a1 (Return _) = a1
interleave (Return _) a2 = a2
```

Outline

```
main :: [Response] -> [Request]
```

```
main :: [Response] -> [Request]

data Request =
    Look Direction
  | ReadLine
  | Fire Direction
  | WriteLine String

data Response =
    Image Picture      -- Response for Look
  | ChatLine String    -- Response for Read
  | Succeeded Bool     -- Response for Write
```

Тут есть недостаток в дизайне типов.

Годный тип для взаимодействия

```
data Interaction next =
    Look Direction (Image -> next)
  | Fire Direction next
  | ReadLine (String -> next)
  | WriteLine String (Bool -> next)
  -- deriving (Functor)
```

```
instance Functor Interaction where
    fmap f (Look dir g)    = Look dir (f . g)
    fmap f (Fire dir x)    = Fire dir (f x)
    fmap f (ReadLine g)    = ReadLine (f . g)
    fmap f (WriteLine s g) = WriteLine s (f . g)
```

```
type Program = Free Interaction

easyToAnger = Free $ ReadLine $ \s -> case s of
    "No" -> Free $ Fire Forward $
            Free $ WriteLine "Выкуси!"
                             (\_ -> easyToAnger)
    _    -> easyToAnger
```

Интерпретируем

```
interpret :: Program r -> Game r
interpret prog = case prog of
    Free (Look dir g) -> do
        img <- collectImage dir
        interpret (g img)
    Free (Fire dir next) -> do
        sendBullet dir
        interpret next
    Free (ReadLine g) -> do
        str <- getChatLine
        interpret (g str)
    Free (WriteLine s g) ->
        putChatLine s
        interpret (g True)
    Pure r -> return r
```

Немного сахара (1/2)

```
look :: Direction -> Program Image
look dir = liftF (Look dir id)

fire :: Direction -> Program ()
fire dir = liftF (Fire dir ())

readLine :: Program String
readLine = liftF (ReadLine id)

writeLine :: String -> Program Bool
writeLine s = liftF (WriteLine s id)
```

Немного сахара (2/2)

```
easyToAnger :: Program a
easyToAnger = forever $ do
    str <- readLine
    when (str == "No") $ do
        fire Forward
        -- Ignore the Bool returned by writeLine
        _ <- writeLine "Take that!"
        return ()
```

Конец.

Ссылки I

Gabriel Gonzalez

Why free monads matter

ссылка