Декларативное программирование

Семинар №13, группа 22215

Завьялов А.А.

28 ноября 2022 г.

Кафедра систем информатики ФИТ НГУ

Вычислительные эффекты

Что такое

Если функция не просто переводит одно значение в другое, но *делает что-то* ещё или *по-особому* работает, говорят, что она содержит некоторый **вычислительный эффект**

Какие бывают

- Частичность
- Недетерминированность
  - Нестабильность
  - Множественность
- Побочный эффект

Вычислительные эффекты

Что такое

Если функция не просто переводит одно значение в другое, но *делает что-то* ещё или *по-особому* работает, говорят, что она содержит некоторый **вычислительный эффект**

Какие бывают

- Частичность – `Maybe`
- Недетерминированность
  - Нестабильность
  - Множественность
- Побочный эффект

Вычислительные эффекты

Что такое

Если функция не просто переводит одно значение в другое, но *делает что-то* ещё или *по-особому* работает, говорят, что она содержит некоторый **вычислительный эффект**

Какие бывают

- Частичность – `Maybe`
- Недетерминированность
  - Нестабильность
  - Множественность – `[]`
- Побочный эффект

Вычислительные эффекты

Что такое

Если функция не просто переводит одно значение в другое, но *делает что-то* ещё или *по-особому* работает, говорят, что она содержит некоторый **вычислительный эффект**

Какие бывают

- Частичность – `Maybe` (и не только)
- Недетерминированность
  - Нестабильность
  - Множественность – `[]` (и не только)
- Побочный эффект

Вычислительные эффекты

Что такое

Если функция не просто переводит одно значение в другое, но *делает что-то* ещё или *по-особому* работает, говорят, что она содержит некоторый **вычислительный эффект**

Какие бывают

- Частичность – `Maybe` (и не только)
- Недетерминированность
  - Нестабильность
  - Множественность – `[]` (и не только)
- Побочный эффект

Монада списка и динамическое программирование

Определите функцию `permutations :: [a] -> [[a]]`, которая возвращает все перестановки, которые можно получить из данного списка, в любом порядке.

Монада списка и динамическое программирование

Определите функцию `permutations :: [a] -> [[a]]`, которая возвращает все перестановки, которые можно получить из данного списка, в любом порядке.

Перестановки разной степени сложности

- перестановки пустого списка – пустой список;
- перестановки одноэлементного списка – тот же самый список;
- перестановки списка `[x, y]` – список `[[x, y], [y,x]]`;
- перестановки списка `[x1, x2, ..., xn]` – перестановки `[x2, ..., xn]` с добавленным в начало `x1`, а также перестановки `[x1, x3, ..., xn]` с добавленным в начало `x2` и т.д...

Сопоставление с образцом в монадах и MonadFail

Переводчик на собачий язык

```
dogefier ::
  Maybe String -> Maybe String
dogefier (Just "hello") =
  Just $ "henlo"
```

Что выведет интерпретатор?

```
> dogefier $ return "hello"
> dogefier $ return "привет"
```

Сопоставление с образцом в монадах и MonadFail

Переводчик на собачий язык

```
dogefier ::
  Maybe String -> Maybe String
dogefier (Just "hello") =
  Just $ "henlo"
```

Что выведет интерпретатор?

```
> dogefier $ return "hello"
> dogefier $ return "привет"
```

Попробуем исправить

Метод `fail` и `do`-нотация

Выражения вида

```
do { Just x1 <- action1
   ; x2      <- action2
   ; mk_action3 x1 x2 }
```

Превращаются в

```
action1 >>= f
where f (Just x1) =
  do { x2 <- action2
     ; mk_action3 x1 x2 }

     f _ = fail "..."
```

- Работает для всех монад, реализующих класс типов `MonadFail` (содержит метод `fail`)
- `fail :: Maybe a` ≡ `Nothing`
- `fail :: [a]` ≡ `[]`

Глубокий смысл `fail` для монады списка

Выражение

```
evens =
  [x |
   x <- [1..],
   x `mod` 2 == 0]
```

Эквивалентно следующему

```
evens = do
  x <- [1..]
  True <- return x `mod` 2 == 0
  return x
```

- Позволяет избавляться от ненужных значений;
- `guard` делает похожую работу, но:
  - мы пока не знаем, как работает `guard`;
  - мы уже знаем, как работает паттерн-матчинг и `fail`.

Эффект нестабильности

Если ваша функция `f :: a -> b` возвращает разные значения при одинаковом значении аргумента, значит она зависит от скрытого параметра.

Эффект нестабильности

Если ваша функция `f :: a -> b` возвращает разные значения при одинаковом значении аргумента, значит она зависит от скрытого параметра.

Чистые функции не зависят от скрытых изменяющихся переменных. Нестабильность можно выразить, сделав зависимость явной:

```
f' :: a -> r -> b
```

Эффект нестабильности

Если ваша функция `f :: a -> b` возвращает разные значения при одинаковом значении аргумента, значит она зависит от скрытого параметра.

Чистые функции не зависят от скрытых изменяющихся переменных. Нестабильность можно выразить, сделав зависимость явной:

```
f' :: a -> r -> b
```

Можем дать `e -> b` псевдоним:

```
type Reader r b = r -> b
f' :: a -> Reader r b
```

Монада Reader

Определение

```haskell
newtype Reader r a = Reader { runReader :: r -> a }

instance Monad (Reader r) where
  return a = Reader $ \_ -> a
  -- Передаёт окружение в оба вычисления
  m >>= k = Reader $ \r -> let v = runReader m r
                           in runReader (k v) r
```

Полезные функции

- `ask = Reader id :: Reader a a`
- `asks f = Reader f :: (r -> a) -> Reader r a`
- `local :: (r -> r) -> Reader r a -> Reader r a`
  `local f m = Reader $ runReader m . f`

Побочный эффект

Если ваша функция `f :: a -> b` не только возвращает значение, но и делает что-то ещё (например, изменяет глобальную переменную), значит, она имеет побочный эффект.

Побочный эффект

Если ваша функция `f :: a -> b` не только возвращает значение, но и делает что-то ещё (например, изменяет глобальную переменную), значит, она имеет побочный эффект.

Чистые функции не могут сделать ничего, кроме подсчета единственного возвращаемого значения. Побочный эффект можно выразить, сделав побочный эффект частью результата функции:

```
f' :: a -> (b, s)
```

Побочный эффект

Если ваша функция `f :: a -> b` не только возвращает значение, но и делает что-то ещё (например, изменяет глобальную переменную), значит, она имеет побочный эффект.

Чистые функции не могут сделать ничего, кроме подсчета единственного возвращаемого значения. Побочный эффект можно выразить, сделав побочный эффект частью результата функции:

```
f' :: a -> (b, s)
```

Можем сделать `(b,s)` осмысленным типом:

```
newtype Putter s b = Putter (b, s)
f' :: a -> Putter s b
```

Монада `Writer`

Определение

```haskell
newtype Writer w a = Writer { runWriter :: (a, w) }

instance Monoid w => Monad (Writer w) where
  return x = Writer (x, mempty)
  Writer (x,v) >>= f = let (Writer (y, v')) = f x
                       in Writer (y, v `mappend` v')
```

Полезные функции

- `tell :: Monoid w => w -> Writer w ()`
- `execWriter :: Writer w a -> w`

Что почитать

- https://ruhaskell.org/posts/theory/2018/01/10/effects.html
- https://ruhaskell.org/posts/theory/2018/01/18/effects-haskell.html

Q&A

Монада State

```haskell
newtype State s a = State { runState :: s -> (a,s) }

instance Monad (State s) where
  -- return :: a -> State s a
  return x = State $ \s -> (x, s)
  -- (>>=) :: State s a -> (a -> State s b) -> State s b
  State act >>= f = State $ \s ->
    let (a, s') = act s
    in runState (k a) s'
```

Полезные функции

- `get :: State s s`
- `put :: s -> State s ()`
- `modify :: (s -> s) -> State s ()`