Монады — не приговор

Виталий Брагилевский

25 июля 2019, Tver.io Pure Meetup, Тверь, Россия

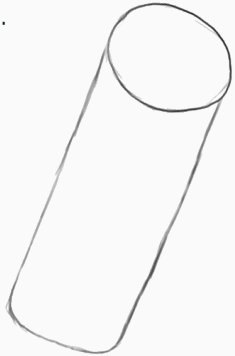
JetBrains (Санкт-Петербург, Россия)

GHC Steering Committee

Монада — это моноид в категории эндофункторов. Ясно?

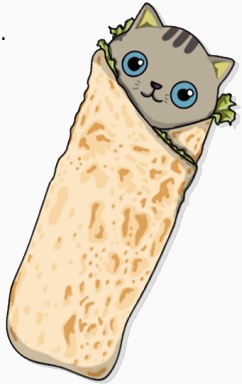
HOW TO WRITE A MONAD TUTORIAL @IMPUREPICS

1.



MENTION THAT THIS ONE
WILL BE DIFFERENT AND EASIER

2.



WRITE THE REST OF
THE DAMN TUTORIAL

https://impurepics.com/posts/2018-04-28-how-to-write-monad-tutorial.html

Этот туториал отличается от других, к тому же он проще

Этот туториал отличается от других, к тому же он проще

1. Простой код в императивном стиле

Этот туториал отличается от других, к тому же он проще

1. Простой код в императивном стиле
2. Тот же код в функциональном стиле

Этот туториал отличается от других, к тому же он проще

1. Простой код в императивном стиле
2. Тот же код в функциональном стиле
3. Монады как обобщение понятия вычисления

Давайте скачаем несколько веб-страниц по списку адресов

```
f = open("urls.txt")
content = read_file(f)
urls = content.split()
for url in urls:
    if url[:7] == "http://":
        response = get_url(url)
        print(url)
        print(response)
```

Давайте скачаем несколько веб-страниц по списку адресов

```
f = open("urls.txt")
content = read_file(f)
urls = content.split()
for url in urls:
    if url[:7] == "http://":
        response = get_url(url)
        print(url)
        print(response)
```

- Императивный стиль

Давайте скачаем несколько веб-страниц по списку адресов

```
f = open("urls.txt")
content = read_file(f)
urls = content.split()
for url in urls:
    if url[:7] == "http://":
        response = get_url(url)
        print(url)
        print(response)
```

- Императивный стиль
- Разноуровневые абстракции

Давайте скачаем несколько веб-страниц по списку адресов

```
f = open("urls.txt")
content = read_file(f)
urls = content.split()
for url in urls:
    if url[:7] == "http://":
        response = get_url(url)
        print(url)
        print(response)
```

- Императивный стиль
- Разноуровневые абстракции
- Высокая насыщенность на единицу кода

Давайте скачаем несколько веб-страниц по списку адресов

```
f = open("urls.txt")
content = read_file(f)
urls = content.split()
for url in urls:
    if url[:7] == "http://":
        response = get_url(url)
        print(url)
        print(response)
```

- Императивный стиль
- Разноуровневые абстракции
- Высокая насыщенность на единицу кода
- Низкая плотность на строчку кода

Давайте скачаем несколько веб-страниц по списку адресов

```
f = open("urls.txt")
content = read_file(f)
urls = content.split()
for url in urls:
    if url[:7] == "http://":
        response = get_url(url)
        print(url)
        print(response)
```

- Императивный стиль
- Разноуровневые абстракции
- Высокая насыщенность на единицу кода
- Низкая плотность на строчку кода
- Много переменных

А теперь то же самое, но в функциональном стиле

А теперь то же самое, но в функциональном стиле

Чтение списка строк из файла

```
urls = split(read_file(open("urls.txt")))
```

А теперь то же самое, но в функциональном стиле

Чтение списка строк из файла

```
urls = split(read_file(open("urls.txt")))
```

```
  split . read_file . open
```

А теперь то же самое, но в функциональном стиле

Чтение списка строк из файла

```
urls = split(read_file(open("urls.txt")))
```

```
  split . read_file . open
```

Фильтрация и обработка списка URL

```
map(process_url, filter(is_http, urls))
```

А теперь то же самое, но в функциональном стиле

Чтение списка строк из файла

```
urls = split(read_file(open("urls.txt")))
```

```
split . read_file . open
```

Фильтрация и обработка списка URL

```
map(process_url, filter(is_http, urls))
```

```
map process_url . filter is_http
```

Соединяем чтение и обработку

```
map(process_url,
    filter(is_http,
           split(read_file(open("urls.txt")))))
```

Соединяем чтение и обработку

```
map(process_url,
    filter(is_http,
           split(read_file(open("urls.txt")))))
```

```
(map process_url
     . filter is_http
     . split . read_file . open) "urls.txt"
```

Реализуем остатки

Предикат для адресов HTTP

```
is_http = (=="http://") . take 7
```

Реализуем остатки

Предикат для адресов HTTP

```
is_http = (=="http://") . take 7
```

Обработка одного адреса

```
process_url =
  \url -> print (url ++ "\n" ++ get_url url)
```

Собираем всё вместе

```
is_http = (=="http://") . take 7

process_url =
  \url -> print (url ++ "\n" ++ get_url url)

(map process_url
     . filter is_http
     . split . read_file . open) "urls.txt"
```

И сравниваем

```
is_http = (=="http://") . take 7
process_url =
  \url -> print (url ++ "\n" ++ get_url url)

(map process_url
     . filter is_http
     . split . read_file . open) "urls.txt"
```

- Функциональный стиль
- Каждая функция работает на одном уровне абстракции
- Низкая насыщенность на единицу кода
- Высокая плотность на строчку кода
- Мало имён переменных

Делаем вывод: есть (как минимум!) два способа записывать вычисления

- последовательность инструкций (плюс ветвления и циклы);
- вызов функции от результата предыдущего вызова функции (плюс условные выражения и рекурсия).

А теперь дадим слово науке

Смотрим на эти два способа записи вычислений и обобщаем!

```
p1();
p2();
p3();
```

```
f3(f2(f1()))
```

Смотрим на эти два способа записи вычислений и обобщаем!

```
p1();
p2();
p3();
```

```
f3(f2(f1()))
```

- Выполняем первый шаг вычисления

Смотрим на эти два способа записи вычислений и обобщаем!

```
p1();
p2();
p3();
```

```
f3(f2(f1()))
```

- Выполняем первый шаг вычисления
- Получаем результат

Смотрим на эти два способа записи вычислений и обобщаем!

```
p1();
p2();
p3();
```

```
f3(f2(f1()))
```

- Выполняем первый шаг вычисления
- Получаем результат
- Используем его для определения второго шага вычисления

Смотрим на эти два способа записи вычислений и обобщаем!

```
p1();
p2();
p3();
```

```
f3(f2(f1()))
```

- Выполняем первый шаг вычисления
- Получаем результат
- Используем его для определения второго шага вычисления
- Выполняем второй шаг вычисления

Смотрим на эти два способа записи вычислений и обобщаем!

```
p1();
p2();
p3();
```

```
f3(f2(f1()))
```

- Выполняем первый шаг вычисления
- Получаем результат
- Используем его для определения второго шага вычисления
- Выполняем второй шаг вычисления
- …

Монады!

Или, точнее, монадические вычисления.

Вводим необходимые технические детали (на типах!)

- Значение типа a

Вводим необходимые технические детали (на типах!)

- Значение типа `a`
- Значение в контексте: `m a`

Вводим необходимые технические детали (на типах!)

- Значение типа `a`
- Значение в контексте: `m a`
- Помещение значения в контекст:

```
pure :: a -> m a
```

Вводим необходимые технические детали (на типах!)

- Значение типа `a`
- Значение в контексте: `m a`
- Помещение значения в контекст:
  ```
  pure :: a -> m a
  ```
- Связывание двух шагов вычисления:
  ```
  (>>=) :: m a -> (a -> m b) -> m b
  ```

Обобщение работает!

```
p1();
p2();
p3();
```

```
p1 >>= (\ _ -> p2)
   >>= (\_ -> p3)
```

Обобщение работает!

```
p1();
p2();
p3();
```

```
p1 >>= (\ _ -> p2)
   >>= (\_ -> p3)
```

```
f3(f2(f1()))
```

```
pure f1 >>= (pure . f2)
        >>= (pure . f3)
```

Монад много, потому что

- контексты бывают разные;
- конкретные реализации `pure` и `(>>=)` зависят от контекста.

Пример: отделяем чистые вычисления от вычислений с эффектами (Haskell)

```
process_url =
  \url -> print (url ++ "\n" ++ get_url url)
```

Пример: отделяем чистые вычисления от вычислений с эффектами (Haskell)

```
process_url =
  \url -> print (url ++ "\n" ++ get_url url)
```

Должно быть

```
(++) :: String -> String -> String
get_url :: String -> IO String
print :: String -> IO ()
```

Пример: отделяем чистые вычисления от вычислений с эффектами (Haskell)

```
process_url =
  \url -> print (url ++ "\n" ++ get_url url)
```

Должно быть

```
(++) :: String -> String -> String
get_url :: String -> IO String
print :: String -> IO ()

process_url =
  \url -> get_url url
          >>= print . (\resp -> url ++ "\n"
                                ++ resp)
```

Нужно это «нормальному» программисту?

Пример: вычисления с возможной ошибкой

Опасно! Стоило бы проверять результаты!

```
v1 = lookup hashmap1 "значение"
v2 = lookup hashmap2 v1
v3 = lookup hashmap3 v2
```

Пример: вычисления с возможной ошибкой

Опасно! Стоило бы проверять результаты!

```
v1 = lookup hashmap1 "значение"
v2 = lookup hashmap2 v1
v3 = lookup hashmap3 v2
```

Безопасно! Поручили проверку функции (>>=)

```
v3 = lookup hashmap1 "значение"
     >>= lookup hashmap2
     >>= lookup hashmap3
```

А это нужно «нормальному» программисту?

Заключаем:

- Языки программирования позволяют вычислять по-разному

Заключаем:

- Языки программирования позволяют вычислять по-разному
- Монады обобщают понятие вычисления

Заключаем:

- Языки программирования позволяют вычислять по-разному
- Монады обобщают понятие вычисления
- Монады можно выносить как средство языка, доступное разработчику, а можно и скрывать за элементами синтаксиса

Q&A

Виталий Брагилевский

vit.bragilevsky@gmail.com

_bravit (ru)

VBragilevsky (en)

bravit111

bravit_about (channel, ru)

Слайды:
bit.ly/bravit-monads