Функциональное программирование. Введение

Косарев Дмитрий

матмех СПбГУ

1 сентября 2022 г.

Дата сборки: 1 сентября 2022 г.

В этих слайдах

Галопом ~~по Европам~~ по основным особенностям *типизированного* функционального программирования

1. Синтаксис вызова функций
2. Алгебраические типы данных
3. Чистые (pure) функции
4. Кортежи
5. Замыкания
6. Хвостовая рекурсия
7. Стандартные функции для списков
8. Вывод типов

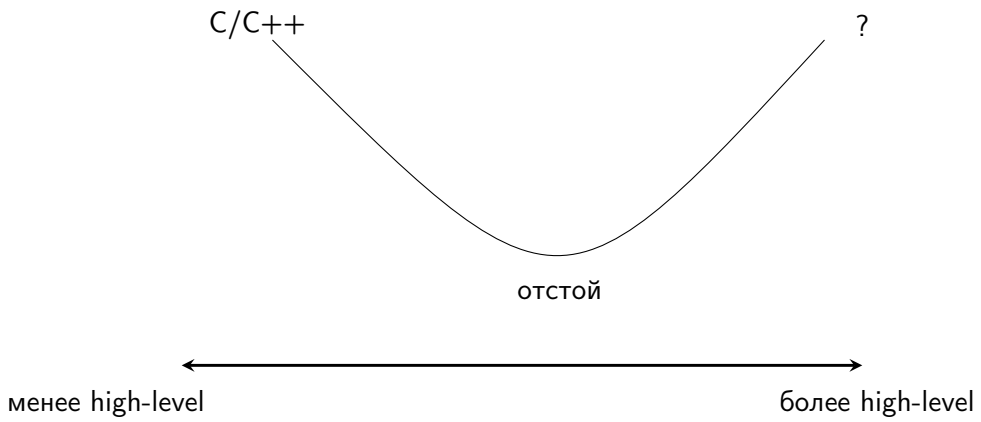
Положение ФП в мире

- Источник идей для других языков
  - Область научных исследований в Computer Science (одна из)
- Отдушина для программистов, которым не достаточно общепринятых подходов
- Практическое применение (непосредственно production)
  - Для "умных" программистов $\downarrow$

Языки для умных vs обычные языки

Доклад от Johnatan Blow к просмотру [1].

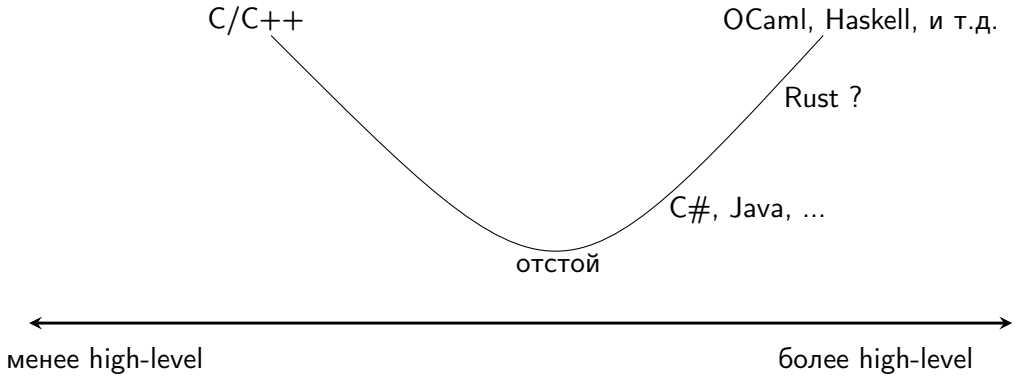
Языки для “умных” программистов vs. языки для “простых” программистов



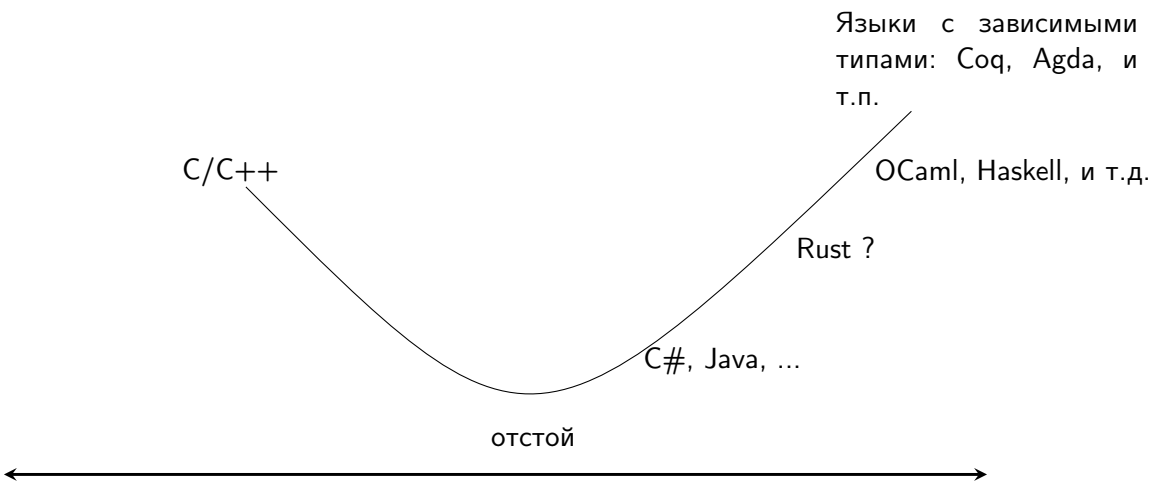
Языки для умных vs языки для умных

Доклад от Johnatan Blow к просмотру [1].

Языки для “умных” программистов vs. языки для “простых” программистов



Хотя на самом деле нужно рисовать вот так



Оглавление

Другой синтаксис вызова функций

В наследниках Си:

```
f(x, g2(y1, y2), z)
```

Сигнатуры:

```
float f(int x, int y, int z)
```

В OCaml (и некотором другом ФП):

```
f x (g2 y1 y2) z
```

Сигнатуры:

```
f : int → int → int → float
```

Такой вид синтаксиса называется *каррированием*

Но можно и без каррирования (как в Си):

```
f(x, g2 y1 y2, z)
```

...

```
f: int * int * int → float
g: int → int → int
```

Каковы явные преимущества каррированных функций?

Можно не писать всюду скобки и запятые.

Например, можно описать API из функций `start`, `fin`, `push`, `add`, `mul` и писать код например так:

```
start push 1 push 2 add push 3 mul fin
```

И выражение будет иметь тип **`int`** и вычисляться в 9

Оглавление

Алгебраические типы данных

Алгебраические типы данных – это скрещивание enum'ов и структур из Си.

Пример: список значений типа `'a`

Связный список – это структура данных, которая представляет собой *или* пустой список (nil), *или* элемент списка (cons), который *состоит из* элемента типа `'a`, хранящегося в списке, *и* ещё одного списка (хвоста).

```
type 'a list  =
  | Nil
  | Cons of 'a * 'a list
```

(Полиморфный) тип `'a list` списков значений типа `'a`

`Nil` и `Cons` – *конструкторы* типа `'a list`

У конструктора `Nil` нет аргументов.

У конструктора `Cons` два аргумента с типами `'a` и `'a list`.

Пример: (односвязные) списки с синтаксический сахар для них

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list
```

Вот так будете писать большинство своих типов данных

```
Nil : 'a list
Cons (1, Nil) : int list

Cons (2, Cons (1, Nil)) : int list
```

```
type 'a list =
  | []
  | (::) of 'a * 'a list
```

Здесь специально выбранный конструктор, чтобы он был инфиксным и право ассоциативным

```
[]       : 'a list
1::[]    : int list
[1]      : int list
1::2::[] : int list
[1;2]    : int list
```

Пример: тип option

```
type 'a option =
  | None
  | Some of 'a
```

Либо нет значения (**None**), либо какое-то есть (**Some**)

Аналог nullptr, только его нельзя случайно разименовать[a]

[a] См. Tony Hoare "Null References: The Billion Dollar Mistake"

```
let run_on_two x y f =
  match x with
  | None → ()
  | Some x →
      match y with
      | None → ()
      | Some y → f x y

(* somewhere in .mli file *)
val run_on_two:
  'a option →
  'b option →
  ('a → 'b → unit) →
  unit
```

Пример: тип bool и “Boolean blindness“

```
(* Встроенный в OCaml *)
true : bool

(* Но можно описать свой *)
type boolean = True | False

(* И потом использовать *)
match ... with
| true →
| false →

if (* condition: bool *)
then ...
else ...
```

```
val is_admin : user → bool
val is_red : node → true
val list_filter: ('a → bool) →
                 'a list → 'a list

(*  Лучше как ниже *)
type role = Admin | User
val get_role : user → role
val color = Red | Black
val get_color : node → color

type filter = Keep | Remove
val list_filter: ('a → filter) →
                 'a list → 'a list
```

Почему алгебраические типы данных (язык Hope, 1980) называются "алгебраическими"?

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection):
  существуют такие s_i^k, что
  $s_i^k(C_k(x_{k_1}, ..., x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness):
  если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity):
  если
  $C_{ij}(x_1, ..., x_{n_{ij}}) = C_{ij}(y_1, ..., y_{n_{ij}})$,
  то $x_k = y_k$
- Полнота (exhaustiveness):
  если x алгебраического типа, то
  $\exists i, n\colon\ x = C_i(y_1, ..., y_n)$

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection): существуют такие s_i^k, что $s_i^k(C_k(x_{k_1}, ..., x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness): если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity): если $C_{ij}(x_1, ..., x_{n_{ij}}) = C_{ij}(y_1, ..., y_{n_{ij}})$, то $x_k = y_k$
- Полнота (exhaustiveness): если x алгебраического типа, то $\exists i, n\colon x = C_i(y_1, ..., y_n)$

Доступ к аргументам можно осуществлять с помощью па́ттерн-мэтчинга

```
let rec somefunc xs =
  match xs with
  | [] →
     (* no argument of empty list*)
     ...
  | h :: tl →
     ... h ... tl ... h ...
```

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection): существуют такие s_i^k, что $s_i^k(C_k(x_{k_1}, ..., x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness): если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity): если $C_{ij}(x_1, ..., x_{n_{ij}}) = C_{ij}(y_1, ..., y_{n_{ij}})$, то $x_k = y_k$
- Полнота (exhaustiveness): если x алгебраического типа, то $\exists i, n\colon\ x = C_i(y_1, ..., y_n)$

Дизъюнктивность: если значения начинаются с разных конструкторов, то они не равны

```
let rec neq_lists xs ys =
  match (xs,ys) with
  | [], _::_ → true
  | _::_, [] → true
  | x::tl1, y::tl2 →
      (x <> y) ||
      neq_lists tl1 tl2
  | [],[] → false
```

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection): существуют такие s_i^k, что $s_i^k(C_k(x_{k_1}, ..., x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness): если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity): если $C_{ij}(x_1, ..., x_{n_{ij}}) = C_{ij}(y_1, ..., y_{n_{ij}})$, то $x_k = y_k$
- Полнота (exhaustiveness): если x алгебраического типа, то $\exists i, n\colon\ x = C_i(y_1, ..., y_n)$

Инъективность: если значения равны, то начинаются с одного и того же конструктора, и аргументы равно попарно.

```
let rec eq_lists xs ys =
  match (xs,ys) with
  | [],[] → true
  | [], _ :: _ → false
  | _ :: _, [] → false
  | x::tl1, y :: tl2 →
      (x = y) &&
      eq_lists tl1 tl2
```

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection): существуют такие s_i^k, что $s_i^k(C_k(x_{k_1}, ..., x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness): если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity): если $C_{ij}(x_1, ..., x_{n_{ij}}) = C_{ij}(y_1, ..., y_{n_{ij}})$, то $x_k = y_k$
- Полнота (exhaustiveness): если x алгебраического типа, то $\exists i, n\colon\ x = C_i(y_1, ..., y_n)$

Полнота: значения алгебраического типа всегда начинаются только с тех конструкторов, которые перечислены в типе

```
let rec even_length xs =
  match xs with
  | [] → true
  | _ :: _ :: tl → even_length tl

(* Warning 8: this pattern-matching
  is not exhaustive.
  Here is an example of a case
  that is not matched:
  _::[]
*)
```

Оглавление

Чистые (pure) функции

Определение

Чистая функция обладает следующими двумя свойствами

- Детерминированная: результат зависит только от аргументов
  - Нельзя использовать глобальные изменяемые данные
  - Нельзя запросить дату на компьютере и в зависимости от неё, что-то делать
- В процессе работы не совершающая “побочных эффектов”
  - Нельзя печатать, что-то на консоль
  - Вызывать внутри себя ”нечистые” функции

Замечания:

- Это свойство *функций*, а не языка программирования
- Их проще отлаживать, некоторые оптимизации (например, inlining) становятся чаще применимы
- Если до этого программировали только с изменяемым состоянием, то перестроиться может быть сложно.

Использование вложенных функций

```
let sum_list: int list → int =
  let rec helper acc xs =
    match xs with
    | [] → acc
    | x::xs → helper (acc+x) xs
  in
  helper 0
```

Здесь в теле функции `sum_list` объявлена функция `helper` с дополнительным параметром-аккумулятором.

Компилятор достаточно умный, чтобы не создавать новую функцию `helper` на каждый вызов `sum_list`.

Использование вложенных функций

```
let sum_list: int list → int =
  let rec helper acc xs =
    match xs with
    | [] → acc
    | x::xs → helper (acc+x) xs
  in
  helper 0
```

```
let sum_list (zs: int list) : int =
  let rec helper acc xs =
    match xs with
    | [] → acc
    | x::xs → helper (acc+x) xs
  in
  helper 0 zs
```

Здесь в теле функции sum_list объявлена функция helper с дополнительным параметром-аккумулятором.

Компилятор достаточно умный, чтобы не создавать новую функцию helper на каждый вызов sum_list.

N.B. η-конверсия при вызове helper корректна, т.е.

```
sum_list xs = helper 0 xs
      ⇕
sum_list = helper 0
```

Оглавление

Встроенные в синтаксис кортежи (n-ки, tuples)

```
let triple: (int, string, float) =
(1, "two", 3.0)
```

Количество элементов в кортежах не ограничено.

Доступ к элементам кортежа

```
let (n, _, _) = triple in
assert (n = 1)
```

А в общем виде с помощью *сопоставления с образцом (pattern matching)* (будет ниже)

По сути, когда вызываем функции в стиле Си, мы передаем не n аргументов, а кортеж из n аргументов

```
f: int * int * int → float
...
let result = f triple in
...
```

Оглавление

Замыкания

Задача

Дана функция make с типом a → b → c, и значение с типом a. Хочется, на отдавая значение типа a, отдать наружу функцию b → c, которая будет строить значения типа c из значений типа b.

Стандартный подход из мира ООП

```
class CreatorCCommand {
  a  _a;
public:
  CreatorCCommand(a a)
    : _a(a) {}
  c execute(b b) {
    return make(_a, b);
  }
};
```

Замыкания

Задача

Дана функция make с типом a → b → c, и значение с типом a. Хочется, на отдавая значение типа a, отдать наружу функцию b → c, которая будет строить значения типа c из значений типа b.

Стандартный подход из мира ООП

```
class CreatorCCommand {
  a _a;
public:
  CreatorCCommand(a a)
    : _a(a) {}
  c execute(b b) {
    return make(_a, b);
  }
};
```

```
let make : a → b → c = fun a b → ...
let make2 : b → c =
  let arg : a = ... in
  make arg
```

Из функции make2 вернули частично примененную функцию, а аргумент хранится в *замыкании (closure)*

Оглавление

Рекурсия предпочтительна, циклов и присваиваний не пишут

```
(* make : 'a → int → 'a list *)
let rec make x n =
  if n<1 then []
  else x :: (make x (n-1))
```

Обычная рекурсивная функция: строим список от головы к хвосту.

Рекурсия предпочтительна, циклов и присваиваний не пишут

```
(* make : 'a → int → 'a list *)
let rec make x n =
  if n<1 then []
  else x :: (make x (n-1))

(* make2 : 'a → int → 'a list *)
let make2 x n =
  let rec helper acc n =
    if n<1 then acc
    else helper (x :: acc) (n-1)
  in
  helper [] n
```

Обычная рекурсивная функция: строим список от головы к хвосту.

Хвостовая рекурсия получается, если результат функции

- либо значение без рекурсивного вызова
- либо вызов той же функции с какими-то аргументами.

Хвостовая рекурсия использует константное количество стека, и поэтому примерно эквивалентна циклу.

Оглавление

Функция List.map

```
(* OCaml *)
let rec map f ys =
  match ys with
  | [] → []
  | x::xs → (f x) :: (map f xs)

let rez = map func xs

# map ((+)1) [1;2;3;4];;
- : int list = [2; 3; 4; 5]
```

```
// C#
var ys = new List<...>();
foreach (var x in xs)
  ys.Add(func(x));

// C#+LINQ
ys.Select(x ⇒ func(x))
ys.Aggregate(0, (acc, x) ⇒ x + acc)
```

Функция List.fold_left

```
(* OCaml *)
let rec fold_left f acc ys =
  match ys with
  | [] → acc
  | x::xs → fold_left f (f acc x) xs

# fold_left (+) 0 [1;2;3;4];;
- : int = 10
# fold_left (^) "0" ["1";"2";"3"];;
- : string = "0123"

# fold_left (^) "0" ["1";"2";"3"];;
- : string = "0123"
```

```
// C#
var acc = init;
foreach (var x in xs)
  acc = f(acc, x);

// C# + LINQ
xs.Aggregate(0,
  (acc, x) ⇒ x + acc)
```

Оглавление

Автоматический вывод типов. Пример 1

```
let s f g x = f x (g x)
```

Вывод:

- `f : 'a → 'b → 'c`
- `g : 'd → 'e`
- `s : ('a → 'b → 'c) → ('d → 'e) → 'x → 'r`
- `'c ~ 'r` т.к. результат `f` – это результат `s`
- `'x ~ 'a` т.к. 1й аргумент `f` – это 3й аргумент `s`
- `'x ~ 'd` аналогично для `g`
- `'e ~ 'b` т.к. результат `g` – это 2й аргумент `s`
- Итого:
  `s : ('a → 'b → 'c) → ('a → 'b) → 'a → 'c`

Автоматический вывод типов. Пример 2

```
let rec map f ys =
  match ys with
  | [] → []
  | x::xs → (f x) :: (map f xs)
```

Трассировка вывода типа:

- `map : 'f → 'ys → 'res`
- `ys : 'y list` и `'y list ~ 'ys`
- `rez : 'r list`
- `'f ~ 'a → 'b` т.к. `f` применяется к какому-то аргументу как функция
- `x: 'y` и `xs: 'y list`
- `'y ~ 'a` т.к. `f` применяется к `x`
- `'r ~ 'b` т.к. результат `f x` складывается в список, который храните значения типа `'r`
- Итого: `map : ('a → 'b) → 'a list → 'b list`

Автоматический вывод типов. Пример 3. (1/2)

Что может пойти не так?

```
let rec map: ('a → 'b) → 'a list → 'b list = fun f ys →
  match ys with
  | [] → []
  | x::xs → x :: (f x) :: (map f xs)
```

Несмотря на то, что приписали тип, вывелось не то, что написали:

```
val map : ('b → 'b) → 'b list → 'b list = <fun>
```

Автоматический вывод типов. Пример 3. (1/2)

Что может пойти не так?

```
let rec map: ('a → 'b) → 'a list → 'b list = fun f ys →
  match ys with
  | [] → []
  | x::xs → x :: (f x) :: (map f xs)
```

Несмотря на то, что приписали тип, вывелось не то, что написали:

```
val map : ('b → 'b) → 'b list → 'b list = <fun>
```

Автоматический вывод типов. Пример 3. (1/2)

Что может пойти не так?

```
let rec map: ('a → 'b) → 'a list → 'b list = fun f ys →
  match ys with
  | [] → []
  | x::xs → x :: (f x) :: (map f xs)
```

Несмотря на то, что приписали тип, вывелось не то, что написали:

```
val map : ('b → 'b) → 'b list → 'b list = <fun>
```

Лайфхак 1:

```
let rec map: 'a 'b . ('a → 'b) → 'a list → 'b list = fun f ys →
  match ys with
  | [] → []
  | x::xs → x :: (f x) :: (map f xs)
(* Error: This definition has type 'c. ('c → 'c) → 'c list → 'c list
     which is less general than 'a 'b. ('a → 'b) → 'a list → 'b list
```

Автоматический вывод типов. Пример 3. (2/2)

Лайфхак 2:

```
(* filename.ml *)
let rec map = fun f ys →
  match ys with
  | [] → []
  | x::xs → x :: (f x) :: (map f xs)

(* filename.mli *)
val map: ('a → 'b) → 'a list → 'b list
```

Файлы интерфейса (signature)

- Объявления значений с типами
- Объявления модулей и типов модулей
- Документация

Файлы реализации (structure)

- Реализация значений
- Реализация модулей
- Сигнатуры типов модулей
- и т.д.

Каким должен быть вывод типов?

- Он должен завершаться (что на самом деле не так просто [4])
- Должен работать быстро
  - От компилятора в целом это тоже требуется

Каким должен быть вывод типов?

- Он должен завершаться (что на самом деле не так просто [4])
- Должен работать быстро
  - От компилятора в целом это тоже требуется
- У выражения не должно быть можно вывести два типа непохожих друг на друга
  - Другими словами: если можно выражению приписать два типа `t1` и `t2`, то должен существовать третий тип `t`, такой что первые два являются его специализацией

```
(int → string) → int list → string list  (* t1 *)
(bool → int) → bool list → int list      (* t2 *)
('a → 'b) → 'a list → 'b list            (* t  *)
```

  - И это требование **сложно** исполнить!

Итоги

Новые понятия

1. каррированные функции
2. вложенные функции
3. сопоставление с образцом и wildcard
4. хвостовая рекурсия
5. автоматический вывод типов
6. разделение на файлы интерфейса и реализации

Ссылки I


Предотвращая коллапс цивилизации (Preventing the Collapse of Civilization), 2019 (in English)
Jonathan Blow
YouTube


Шпаргалки по синтаксису OCaml
OCamlPro
4 PDF online


Книга Real World OCaml издания 2.0 (есть русский перевод издания 1.0)
online


Java Generics are Turing Complete
Radu Grigore
online

Ссылки II


A Brief History of Algebraic Data Types
Li-yao Xia
online