О функциональном программировании

ИТМО, 5 сентября 2025 года

История вопроса

- Моисей Шейнфинкель, кобминаторы, 1924.
- Алонзо Чёрч (конец 1920х — начало 1930х), лямбда-исчисление: попытка построить логику, в которой вызов функции — основание языка. Парадоксы, добавление типов, просто типизированное лямбда исчисление.
- Помните? $2 := \lambda f.\lambda x.f\ (f\ x)$, и тогда $2\ 2 =_\beta 4$
- Просто-типизированное лямбда исчисление (1940):

$$\lambda f.\lambda x.f\ (f\ x) : (\alpha \to \alpha) \to (\alpha \to \alpha)$$

- Отсюда изоморфизм Карри-Ховарда:

λ-исчисление	логическое исчисление
Лямбда выражение	доказательство
Тип выражения	высказывание

  То есть, $\lambda x.x$ доказывает то, что $\alpha \to \alpha$.
- Имея формальный алгоритм (лямбда исчисление) и математический фундамент для него, большое искушение — построить язык программирования на этой основе.

Динамически и статически типизированные функциональные языки

- Lisp — один из первых языков высокого уровня (1958), динамическая типизация.
- Типовая система Хиндли-Милнера (разрешимая часть интуиционистского ИП 2 порядка) и язык ML (“Meta Language”). Исходно — для написания тактик
- Xavier Leroy, “Top-level loop considered harmful”. Программа живёт в некотором мире (привет, Jupyter Notebook, также промпты-транскрипты), из него можно что-то экстрактировать — против программы, записанной в текстовом файле в законченном виде.
- Caml (Язык категорной абстрактной машины)

Прежде чем скажем, что такое Ф.П.: а что такое функция?

- Определение 1: выделенная часть программы, с соответствующим интерфейсом (подробности опущены).
- Определение 2: математическое определение:

$$\langle D, C, \Gamma \rangle, \text{ где } \Gamma \subseteq D \times C, \qquad \forall d, c_1, c_2 . \langle d, c_1 \rangle \in \Gamma \,\&\, \langle d, c_2 \rangle \in \Gamma \to c_1 = c_2$$

  Скажем, возведение в квадрат: $\langle \mathbb{R}, \mathbb{R}, \{\langle p, p^2 \rangle \mid p \in \mathbb{R}\} \rangle$
- Функции в смысле 1, которые соответствуют определению 2 — “чистые”:

```
int sq(int x) { return x*x; }
```

- Функции в смысле 1, которые не соответствуют определению 2 — “с побочным эффектом”:

```
int seq_no = 0;
int get_seq_no() { return seq_no++; }
printf("Hello, world!\n");      // Изменяет мир
scanf("%d", &v);                // Также изменяет мир
```

Что такое функциональные языки (некоторые общие соображения)

- Функции в Ф.Я. — значения первого (высшего?) сорта (“first class values”), то есть полноценные значения. Например, функцию возможно задать, не указывая имени: $\lambda x.x^2$. Это базовый критерий, но есть и дополнительные.
- Дизайн языка — как правильно ограничить несущественное, чтобы получить важное. Функциональные языки часто предполагают запрет (сильное ограничение) на разрушающее присваивание и побочные эффекты.
- Что это даёт:
  - “Прозрачность по ссылкам” — ссылка на разделяемый объект неотличима от ссылки на его копию.
  - Возможность для ленивых вычислений/оптимизаций/предсказуемость поведения.
- Кроме того, дополнительные характерные возможности, реализация которых облегчается (но которые встречаются и в других языках):
  - Алгебраические типы, сопоставление с образцом, списки, функции высших порядков — базовые конструкции языка.
  - Формализация типовой системы как логического исчисления.
  - Автоматическая сборка мусора.

Haskell

Haskell (в честь Хаскеля Карри), сперва Miranda/Hope (David Turner), Haskell 98:

- Типовая система Хиндли-Милнера
- Прозрачность по ссылкам, запрет разрушающего присваивания и ограничение побочных эффектов.
- Исключения запрещены — хотя впервые появились как раз в Caml, как функциональный goto.
- Форматирование как синтаксическая конструкция.
- Ленивые вычисления.
- Функции высших порядков, list comprehensions и т.п.

Типовая система

Изоморфизм Карри-Ховарда: тип (в лямбда исчислении) — логическое выражение в некотором исчислении. То есть, тип данной функции:

```
compose f g = \x -> f (g x)
```

Соответствует некоторому логическому утверждению:

$$\texttt{compose} : \forall\alpha.\forall\beta.\forall\gamma.(\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

Зачем? 1. упрощение разработки компилятора; 2. регулярность (отсюда: выразительная сила языка).

Запрет на разрушающее присваивание

- Однократное можно: `let x = 5 in x * x`
- Как переписать код в таком стиле поясним на примере:

```
def fib(n):
    a = 1
    b = 1
    for id in range(n):
        (a,b) = (a+b,a)
    return b
```

- Перепишем это с использованием рекурсии:

```
fib' a b n =
    if n > 0
        then fib' (a+b) a (n-1)
        else b

fib n = fib' 1 1 n
```

Синтаксически значимое форматирование

Известно ещё 60х годов. Фортран: первые пять позиций перфокарты — для метки. Позиции с 7 по 72 — текст оператора, с 73 — комментарий.
Зачем: отлаживали?

```
if (x >= 0)
   printf("x >= 0");
else
   printf("clearing negative x");
   x = 0;
```

Ленивые вычисления

- На теоретическом уровне: Normal reduction order для лямбда-выражений
- Ближе к реализации: СЗНФ, “не вычисляем значение, если оно не нужно”
- `LazyValue V = Just V | Closure (() -> V)`

Функции часто возможно задать при помощи ряда: $f(x) = \sum_n a_n \cdot x^n$

```
series f = f : repeat 0                  -- [f,0,0,0,..], т.е. f = f + 0*x + 0*x^2 + ...
summ x [] = 0
summ x (p0:ps) = p0 + x * summ x ps

instance (Num a, Eq a) => Num [a] where
   fromInteger c = series(fromInteger c)
   negate (f:ft) = -f : -ft
   (f:ft) + (g:gt) = f+g : ft+gt

instance (Fractional a, Eq a) => Fractional [a] where
   (f:ft) / (g:gt) = qs where qs = f/g : series(1/g)*(ft-qs*gt)

integral fs = 0 : zipWith (/) fs [1..]  -- p1+p2*x+... = (c+p1*x/1+p2*x^2/2+...)'

sin_s = integral cos_s
cos_s = 1 - integral sin_s

main = do print $ (fromRational $ summ (3.1415926/4) $ take 15 sin_s :: Double)
```

https://www.cs.dartmouth.edu/~doug/powser.html

Применение ленивых вычислений

- Заметим, что if всегда ленив:

```
let fact n = if n > 0 then fact (n-1)*n else 1
```

- Конструкции весьма распространены: например, && и ||, также unwrap_or против unwrap_or_else
- Отлаживали непонятную потерю производительности?

```
fn debug(s: String) {
    if errorlevel <= DEBUG { println!("DEBUG: {}", s); }
}
debug(format!("Nice info: {}", complicated_computation()));
```

- Средство модульности:

```
txt = read_json_file()
l = parse_structure(txt)
parse_integer(l!!1)
```

Если всё правильно сделано, функции не прочтут (и не разберут) лишнего.

- Оптимизация: ленивое соединение списков имеет сложность $O(1)$.

Чисто функциональные структуры данных: очередь

- А что с обычными структурами (массивы, очереди, ...)? Chris Okasaki, “Purely functional data structures”
- Циклический массив? С переотведением памяти?

```
X array [SIZE];
int wr, rd;
void push (X x) { array[wr] = x; wr = (wr+1) % SIZE; }
X& take() { X& tmp = &array[rd]; rd = (rd+1) % SIZE; return tmp; }
```

- Двусвязный список?
  Везде нужно исправлять значения (прозрачности по ссылкам нет), всё дорогое...

Чисто функциональная реализация очереди

```
data Queue a = Queue ([a], [a])       -- входной и выходной списки

push :: a -> Queue a -> Queue a
push last (Queue (qi, qo))  = Queue (last : qi, qo)

take :: Queue a -> (Maybe a, Queue a)
take (Queue ([], []))       = (Nothing, Queue ([], []))
take (Queue (qi, fst : qo)) = (Just fst, Queue (qi, qo))
take (Queue (qi, []))       = take (Queue ([], rev qi))
```

Средняя сложность — $\Omega(1)$ (получается не для всех структур, бывает $\Omega(\log n)$).
При этом очередь (почти) бесплатно копируется и прозрачна по ссылкам:

```
u = push (1, Queue([],[]))
(x,u') = take u
-- тут существуют и u, и u'
```

В этом решении есть свои сложности: многократное использование `u` означает многократное обращение списка. Если предполагается частое копирование очереди, может потребоваться более сложная реализация.

Применение вне Хаскеля

В любой ситуации, где сложности с разрушающим присваиванием (например, многопоточное окружение: Lock-free data structures).
Мы не изменяем структуру данных, только что-то достраиваем вокруг (ну и собираем мусор при необходимости), а потом меняем дешёвую или вообще атомарную ссылку на корень структуры.

Дальнейшее развитие

- Хаскель и ФП в целом послужили базой (источником вдохновения) для многих других языков.
- Близкие проекты (Clean, линейные типы данных)
- Усложнение идеи (ещё более сложные логические теории, используемые в качестве логической основы языка):
  - языки для формализации математических доказательств на основании изоморфизма Карри-Ховарада:
    Ocaml — в Coq
    Haskell — в Agda
    Отдельно Lean, Arend
  - языки с зависимыми типами (например, Идрис)
- Выделение отдельных полезных/удобных конструкций, при этом сохранение общего стиля императивных/объектно-ориентированных языков.
  - Питон: генераторы, работа со списками, форматирование
  - Раст

Раст

Раст подходит ко многим идеям из Ф.П. с практической точки зрения (частное удобное полезное применение лучше общего сложного механизма):

- Автоматическое управление памятью через подсчёт ссылок (autoptr/Rust). Метод идейно проще, но менее эффективен.
- Итераторы как (нехранимые) списки — контролируемые пользователем оптимизации, реализуемые в Хаскеле через общий механизм дефорестации и ленивых вычислений.
- Контроль владения (императивная версия прозрачности по ссылкам — компилятор даёт некоторые гарантии по поводу значений).
- Типовая система — попытка широко использовать вывод типов, хотя система не имеет чёткой формализации.
- Алгебраические типы, сопоставление с образцом.