Функциональное программирование

Akhtyamov Pavel

April 30, 2015

1 Деревья

1.1 Деревья общего вида

Деревья используются в структурах данных (к примеру, для двоичного поиска), в синтаксическом анализе, и у них различная структура.

Дерево - элемент типа T + список деревьев типа T.

```
let rec map f = function
| Leaf(x) -> Leaf(f x)
| Node(x, l) -> Node(f x, List.map (map f) l)
```

Fold (правая свертка, раскрывается снизу вверх)

```
let rec fold f i = function
| Leaf(x) -> f i x
| Node(x, l) -> l |> List.fold (fold f) (f i x)
```

Exercise 1.1. Отображение - отрубить листья у дерева.

Example 1.1. Абстрактное синтаксическое дерево

```
type 't Expr =
| Add of 't * Expr * 't Expr
| Sub of 't * Expr * 't Expr
| Mul of 't * Expr * 't Expr
| Div of 't * Expr * 't Expr
| Neg of 't * Expr
| Value of 't

let rec compute = function
| Add(x, y) -> compute x + compute y
| Sub(x, y) -> compute x - compute y
| Mul(x, y) -> compute x * compute y
| Div(x, y) -> compute x / compute y
| Neg(x) -> -compute x
| Value(x) -> x
```

Как построить по списку дерево? Для этого есть разбор выражения в дерево. На вход получить поток, взять столько, сколько смогло сработать (а затем вернуть оставшуюся часть). Возвращается дерево и хвост, который не обработан.

```
let mknode = function
| '+' -> Add
| '-' -> Sub
| '*' -> Mul
| '/' -> Div

let parse_infix expr =
```

```
let rec ff = function
| [] -> failwith "Error"
| h::t when isDigit -> parse
| h::t when h= ' ' -> ff t
| h::t when h='~' -> // negate
        ...
| h::t -> ... // binary
```

Дерево может порождаться функцией (в не чисто функциональных языках).

1.2 Двоичные деревья

Двоичное дерево представляется в виде узла и двух детей.

Есть несколько обходов двоичного дерева:

- Префиксный
- Инфиксный
- Постфиксный

Можно заменить энергичные вычисления на ленивые (см. код).

Двоичное дерво - все элементы меньше — слева, больше — справа

Хвостовая рекурсия для дерева: вроде бы невозможно (нет линейности). Но используется подход продолжения.

```
let plus1 x f = f(x+1)
let times2 x f = f (x*2)
plus1 1 (fun x -> times 2 x (printfn "%d")) // order way used
```

Иногда удобно использовать аккумулятор вкупе с продолжением (для левого поддерева считаем, для правого — запоминаем, что должны вычислить).

Corollary 1.1. *Таким образом, хотя и дереьев нет в библиотеке, они позволяют организовать достаточно интересные и богатые структуры данных.*

2 Погружение в функциональную составляющую

Очередь на списке — это плохо (необходим кольцевой список). Надо придумывать реализацию (Chris Okazaki). Нам надо помещать быстро элемент в очередь и изымать начало. Подойдет реализация на двух стеках. Перестройка амортизационно идет за $O(1)$.

```
let tail(L,R) =
match L with
| [x] -> (List.rev R, [])
| h::t -> (t, R)
```

Рассмотрим ZipperList (двунаправленный список с выделенным текущим элементом, реализация — alt+Tab и машина Тьюринга).

Zipper может рассматриваться не только на списках, но и на других структурах, в частности, на дереве (написать get для последовательности элементов). Далее хотим, чтобы элемент был доступен корню (идея — поддерево будет перекидываться в левый список), т.е идем налево — запоминаем правое поддерево и наоборот. Таким образом, аналог *Zipper* — устройство директорий в проводнике.

Zipper подчиняются дифференцированию: если $T(A,R) = 1 + A * R * R$, то тогда $Z = 2 * A * R$, где $2 = Left|Right$, A — сохраняемый корневой элемент, B — путь.

2.1 Числовые структуры данных

Сложение можно ассоциировать со списками и, к примеру, ординалами. При этом сложение будет выполняться за $O(n)$. Но в десятичной системе числа складывать удобнее. Числовые структуры данных напоминают числа и группировку данных по образцу. Оптимальная позиционная система счисления — по основанию e. Но используют двоичные списки.

```
type 'a blist =
| Nil
| Zero of ('a*'a) blist
| One of 'a * ('a * 'a) blist
```

Nil — 0 One (1, Nil) —1 Zero(One (1,2), Nil) —2, и т.д.

2.2 Замыкания

Рассмотрим функцию вычисления производной:

```
let deriv f (dx: float) =
        fun x -> (f(x + dx) - f(x))/dx
```

Но проблемы в том, что f и dx запомнились. В итоге, в функциональных языках мы можем пользоваться функциями с внутренними состояниями. А когда можно забыть переменную, которую мы захватили? Нужен *Garbage Collector*! Но замыкания достаточно сложно реализуемы.

Как можно пользоваться замыканиями?

В *F#* есть механизм, позволяющий использовать динамическое связывание.

```
let mutable x = 4
let adder y = x + y
adder 1
x <- 3
adder 1
```

Из примера видно, что сохраняется ссылка.

Замыкание возникает при частичном применении функции. При (+) 1 есть ссылка на библиотечную функцию.

Рассмотрим специализацию:

Программа: $P : I \to O$.

Interpreter: $Source \to Input \to Output$

Compiled=$(Interpreter\ Source) : Input \to Output$. На таком подходе *Futamura Projection* возник подход суперкомпляции (лазить внутри кода и оптимизация в нем).

2.3 Генераторы

Файл больше самого большого массива (ибо нам надо последовательный участок памяти). Но ведь можно считать все посимвольно, и возникает понятие генератора (производить последовательность вычислений, возможно, бесконечную). Это можно реализовать на императивно-функциональном языка (взято с *Lisp*).

```
type cell = {mutable content: int}
let new_counter n =
        let x = {content = n} in
        fun () -> (x.content <- x.conten + 1; x.content)
```

Можно использовать *ref*. Счетчик — частный случай генератора.

```
let new_generator fgen init =
        let x = ref init in
        fun () -> (x:= fgen |x; |x)

let fib = new_generator (fun(u,v) -> (u+v, u)) (1,1) // pairs

let map f g = fun() -> f(g())
let fibs = (fib |> map fst)
```

Можно описать функцию *filter*.

С помощью генераторов можно генерировать бесконечные последовательности.

3 Монады

Монады необходимы, к примеру, для организации ввода-вывода (технология была создана в 1991 году).

К примеру, есть куча проверок на *null*.

```
let read() =
    printf ">"
    let s = Console.Readline()
    try
        Some(int(s))
    with
        _ -> None
```

Для сложения двух чисел в таком коде необходимо выполнить много проверок (если надо было бы складывать 10 чисел, то получилось плохо). Хотелось бы ввести число, сложить его, и так далее (отделить смысловой алгоритм от промежуточных проверок).

```
let bind a f =
    if a = None then None
    else f (a.Value)
bind (read()) (fun x -> bind (read()) (fun y -> Some(x + y))
```

Или

```
let (>>=) a f =
    if a = None then None
    else f (a.Value)
read() >>= fun x -> read() >>= fun y -> Some(x + y)
```

Код был приведен к простому виду.

Недетерминированные вычисления: Вася 2000 года рождения, Петя в два, либо в 3 раза старшу, а Лена на год старше Пети. Сколько лет Лене?

```
let (>>=) mA f = List.collect f mA
[10;11] >>= (fun x -> [x*2; x*3]) >>= (fun x -> [x+1])
```

Как видно, что обработка скинута в bind.

Можно ввести операции типа return:

```
let ret x = [x]
let ret' x = x
let fail = []
ret' [10;11] >>= (fun x -> ret' [x*2;x*3]) >>= (fun x -> ret (x + 1))
```

Перейдем к определению монад:

- Для каждого базового типа есть обрамляющий тип M<t>
- Есть операция return: 't -> M<'t>
- Есть операция композиции bind:

  $- >>=: M <' a > \to (a - > M <' b >) \to M <' b >$

Таким образом, монады описывают последовательное применение вычислений, но логика спрятана внутри (можно спрятать всю магию в ";")

Монада MayBe:

```
//type 't option
return x = Some(x)
bind //аналогично первому примеру
```

Монады должна удовлетворять свойствам:

```
return x >>= f ≡ f x
m >>= return ≡ m
(m >>= f) >>=g ≡ m >>= (λx.fx -> g(x)) // ассоциативность выполнения операций
```

Как спрятать это дело от пользователя в F#:

```
let r = nondet {

    let! vasya = [14;15]
    let! petya = [2*vasya; 3*vasya]
    let lena  = petya + 1
    return lena

}
```

Весь bind запрятан в nondet (let' — аналог ret')

```
type NonderBuilder() =

    number b.Return(x) = ret x
    number b.ReturnFrom(x) = ret' x
    number b.Bind(mA, f) = ma >>= f

let nondet = new NondetBuilder()
```

Помимо return и bind, можно описывать другие свойства (можно определить удаление списка)

```
let rec remove x l =

    nondet {
        if l = [] then return []
        else


            if (List.head l) = x then return (List)... //нет else -> реализуется через zero
    }
```

С помощью монад можно решать логические задачи (задача про льва и единорога: можно перебрать все дни).

- Надо получить тройку для вчерашнего дня
- Проверить выражение на правдивость
- Совершить решение:

```
nondet {

    let (l,e,d) = data
    let (l1, e1, d1) = prev (true, true, "sun") (fun(_,_,x) -> x = d) data
    if (realday l false) = l1 && (realday e false) = e1 then return (l,e,d)

}
```

По сути, мы решили задачу, которую можно решать в логической парадиме.

Рассмотрим примеры классических монад:

а) Монады ввода-вывода — состояние ввода и вывода есть пара списков

```
type State = string list * string list
```

```
type IO <'t> = 't*State
let print(t:string) ((I,O):State) = (t, (I, t::O))
read ((I,O):state) // передвижение каретки
```

б) Монады состояния: пример — проход по дереву (тупая реализация — протаскивать глобальную переменную). С помощью state можно протащить все данные внутри:

```
State<'t> = 's -> 't * 's
return x = fun s -> x, s
let (>>=) x f = (fun s0 -> let a, s = x s0; f a s)
let getState = (fun s -> s, s)
let setState s = (fun _ -> (), s)
//do! - отбросить unit - setState пишется с do
```

в) Вычисления с продолжениями (скинуть хвостовую рекурсию в дереве или определить порядок вычислений в ленивом языке) — можно сказать, что не надо возвращаться

```
let (>>=) x f = x f
read >>= double >>= print >>= (fun x -> x)
```

Быстрая сортировка с хвостовой рекурсией:

```
let qsort list =
    let rec sort list cont =
        match list with
        | [] -> cont []
        | x::xs ->
            let l,r =
              List.partition ((>) x ) xs
            sort l (fun ls -> sort r (fun rs -> cont(ls @ (x::rs))))
    sort list (fun x -> x)
```

Но выглядит это не очень красиво. Надо прятать в монаду продолжения:

```
let qsort list =
    let rec sort list =
        cont {
            match list with
            | [] -> return []
            | x::xs ->
              let l,r =
                List.partition ((>) x ) xs
              let! ls = sort l
              let! rs = sort r
              return ls @ (x::rs)
        }
    sort list (fun x -> x)
```

По сути, нехвостовая рекурсия заменена по внешнему виду на хвостовую, но с протаскиванием монады.

Рассмотрим монадические парсеры.

Как решаются задачи разбора текста:

- По тексту строится синтаксическое дерево
- Правила задаются с помощью грамматики

- Возможные подходы:
  - Лексический анализатор -> синтаксический анализатор
  - Руками
  - Монадический парсер (складываются из кусков - fparsec)

```
let pair = parse {
    let! _ = str_ws "("
    let! n1 = int_ws
    let! _ = str_ws ","
    ...
}
let str_ws str = parse {
    do! skipString str
    do! spaces
    return ()
}
let int_ws = parse {
    ...
}
```

3.1 Параллельные и асинхронные вычисления

В функциональных языках можно вычислять что-то параллельно удобным образом. Но, в целом, средства из коробки реализовать сложно (в Haskell, к примеру).

Можно параллелить вручную: ставить Semaphore, Locks, etc. Есть расширения из .NET.

Основные проблемы асинхронных и параллельных вычислений:

- Общая память (решается легко в функциональных языках)
- Инверсия управления (надо вызывать функцию в функцией и т.д.)

В F# есть Asynchronous Workflows:

```
let task1 = async {return 10 + 10}
let task2 = async {return 20 + 20}
Async.RunSynchronously {Asyc.Parallel [task1; task2]}
let map' func items =
    let tasks =
       seq {
          for i in items -> async { return (func i) }
       }
    Async.RunSynchronously (Async.Parallel tasks)
```

В обычных языках код плохой: создавать каждый раз поток. Но в F# есть планировщик потоков, который не создает новые потоки, а раздает потокам задания.

Асинхронность вкладывается в данный контекст:

```
let httpAsync(url:string) =
async {
    let req = WebRequest.Create(url)
    let! resp = req.AsyncGetResponse()
    use stream = resp.GetResponseStream()
    use reader = new StreamReader(stream)
    let! text = Async.SwaitTask(reader.ReadToEndAsync())
    return text
```

```
}
```

Async имеет монадический синтаксис (хороший синтаксис продолжения). Реализация будет короче и проще, чем в C#. Таким образом, решена проблема инверсии.

Асинхронное чтение с диска:

```
let Readfile fn
let goodfiles
let goodsplit
let wc s = s |> goodsplit |> Seq.filter(fun s -> s.Length > 0) |> Seq.length
goodfiles |> Seq.map (fun f -> (f, wc f)) |> Seq.toArray
let ReadfileAsync fn //заменить read на asyncRead
goodfiles
    |> Seq.map (fun f -> async {
        let! s = ReadFileAsync f
        return (f, wc s)
        })
    |> Async.Parallel
    |> Async.RunSyncronously
```

Программа проработала очень быстро.

Проблема: задачи распараллеливаются, если уже есть массив данных. Но в реальной жизни бывает не так: есть сложная система, возникает проблема масштабирования. Можно использовать паттерн Agent.

```
let rec agent = MailboxProcessor.Start(fun inbox ->
    let rec loop() = async {
        let! msg = ...
        return! loop()
    }
    loop()
)
```

Как написать Crawl?

```
let GetLinks s =
let Crawl url =
    async {
        let! s = httpAsync url
        return GetLinks s
    }
let CrawlAgent = MailboxProcessor.Start(fun inbox ->
    let rec loop() = sync {
        let! msg = inbox.Receive()
        let! res = Crawl msg
        res |> Seq.iter (fun x -> CrawlerAgent.Post x)
        return! loop()
    }
    loop()
)
let Dispatcher n f = // параллельный планировщик агентов, который обрабатывает пул сообщений
```

Таким образом, монады дают прекрасный способ описывать параллельные вычисления.

Exercise 3.1. Написать частотный словарь Рунета (можно красиво написать на основе реактивного программирования).

Иногда можно вычислять параллельные задачи на облаке (кластерах). Есть проект {m}brace, который помогает рассмотреть данный вопрос. Для этого есть монада cloud. На кластере есть специальный подход к вычислениям, который воспринимается им при помощи программы. (m-brace.net, briskengine.com). Такой кластер можно развернуть на Microsoft Azure.

Можно это сделать другим образом (при помощи CloudFlow). Здесь использована идея Map-Reduce, которая хорошо используется на облачных серверах.

Внутренние переменные внутри cloud автоматически загружаются в облако (чеерз cloudFlow можно загрузить все вручную).

4 Реактивное функциональное программирование

В прошлом разделе мы написали Crawl, который индексирует веб-сайты. В нем использовался паттерн Agent. Как измерить доброту Интернета? Не очень красивое решение: можно запилить функцию (но функцию надо будет менять).

Есть некоторые механизмы обработки:

- списки; (обработка в памти)
- последовательности; (применять вытаскивание следующего элемента)
- потоки; (вызов функциии на каждом элементе)

```
type Stream<'T> = ('T -> unit) -> unit
```

По сути, данные передаются в поток, а затем применяются функции (причем про input мы не знаем информации).

Поэтому хочется, чтобы Crawler возвращал поток.

Реактивное программирование — парадигма, ориентированнная на потоки данных и распространение изменений, типичный пример — таблицы в Excel. Reactive Manifesto — реактивный стиль в классических языках программирования.

Реактивное функциональное программирование бывает дискретным и непрерывным.

```
type Crawler() =

    let evt = new Event<_>()
    ...

evt.Trigger --- отправляет события
evt.Publish --- подписка (несколько получателей на одно событие).
```

Доброта Твиттера: создаем event, далее создаем listener, который получает данные в виде JSON (работа с которым хороша с помощью NewtonSoft.JSON). В данном случае в FSharpChart дается на вход Event, и она отображается в виде FSharpChart.FastLine

Еще одним примером является естественный контроль движений руки (при помощи Kinect). На каждом шаге передаем фрейм с устройства и контроллер, можно посчитать скорость пальцев или тренажер пальцев при помощи функции Event.Scan (аккумулировать максимум). Еще можно посмотреть при помощи контроллера на 5 фреймов назад и проверять движения руки на жесты.

При помощи Kinect можно получать результаты разных упражнений.

Еще можно использовать реактивное программирование на облаке (каждый кластер обрабатывает данные, а затем их отдает на узел, который складывается в очередь и обрабатывается).

Event — самый простой вид обработки, но что делать, если возникают ошибки? В .NET есть Reactive Extensions:

```
IObserver<T>(OnNext, OnError), IObservable<T>
```

Как можно программировать робота? Сенсоры предоставляют Observable объекты:

```
//init models
let power =  sensor.ToObservable().Select(dist).DistinctUntilChanged()
use l = power.Subscribe(motorL)
use r = power.Subscribe(motorR)
```

На робот можно посылать данные с мобильных устройств (логика на клиенте, на роботе — простая работа): подписываем первую координату на один мотор, вторую — на другой мотор. Создается TrikPad, который позволяет по наклону руки получает данные, которые надо послать на робота.

В итоге, реактивное программирование является хорошим способом создания надежных систем регаирования в реальном времени. Еще реактивный подход позволяет работать с потоками событий с использованием привычных примитивов.

Хорошим примером является обработка событий с мышки в пользовательском интерфейсе, а еще есть хорошее применение в real-time анализе котировок.

5 Реализация функциональных языков программрования

Рассмотрим возможные реализации функциональных языков программирования.

Функциональный язык представляет расширенный функционал относительно λ-исчисления. А здесь есть различия:

- Прямая интерпретация (хороша для энергичных языков и если не гонимся за производительностью языков)
- Редукция графов (изначально есть дерево, а далее идет редукция до необходимого состояния, иногда доходит до графов)
- Использование комбинаторов — комбинаторная редукция
- Код абстрактной машины — на их основе можно построить интерпретатор абстрактной машины (примеры SECD, .NET и т.п.)

```
type Expr =
    | App of Expr*Expr
    | Lam of string*Expr
    | Var of string
    | Cons of string
    | Clo of string*Expr*Env //лямбда с захваченным окружением

and Env = Map<string, Expr>
let text = App(App(Lam("f", Lam("x", App(var...)))))
let rec eval env expr =
match expr with
    | App(e1, e2) ->
       let f1 = eval env e1
       let f2 = eval env e2
       match f1 with
           | Clo(x, ex, env') -> eval(Map.add x f2 env') ex // добавление в окружение
           | Cons(x) -> App(Cons(x), f2)
    | Var(x) -> Map.find x env
    | Cons(x) -> Cons(x)
    | Lam(x, e) -> Clo(x, e, env)

eval Map.empty text
```

В Lisp интерпретатор состоит из eval и apply:

```
| Let of id*expr*expr
| Pfunc of id // примитивная функция
| Op of id*int*expr list // отложенные операции, int --- число нехватаемых выражений
```

Eval:

```
let rec eval exp env =
```

```
    match exp with
    | App(e1, e2) -> apply (eval e1 env) (eval e2 env)
    | Int(n) -> Int(n)
    | Var(x) -> Map.find x env
    | PGunc(f) -> Op(f, arity f, [])
    ...
let apply e1 e2 =
    match e1 with
        | Closure(Lam(v, e), env) -> eval e (Map.add v e2 env)
        | Op(id, n, args) ->
           if n = 1 then (funof id)(args@[e2])
           else Op(id, n01, args@[e2])
```

Таким образом был устроен интерпретатор Lisp.

Возникает вопрос: как работать с рекурсией?

```
letrec fact =
fun x ->
    if x <= 1 then 1
    else x*fact(x-1)
in fact 4
```

Есть другой комбинатор неподвижной точки для энергичных вычислений, но он огромный. Нам надо, чтобы в правой части вместо fact подставилось определение этой функции. Надо придумать let, который подставляет окружение функции (нужно добавить рекурсивное замыкание).

```
eval ...
| LetRec(id, e1, e2) -> eval e2 (Map.add id (RClosure(e1, env, id)) env)
apply ...
| RClosure(Lam(v, e), env, id) -> eval e (Map.add v e2 (Map.add id e1, env)) // закрутка выражений?
```

Как оказывается, letrec не тоже самое, что и let.

Как добавить ленивые вычисления? Сначала вычисляем f, а вычисление x откладываем. Для откладывания вводится операция Susp:

```
|App(e1, e2) -> apply(eval e1 env) (Susp(e2, env))
... some changes with primitive functions
```

Перейдем к рассмотрению реализации при помощи абстрактных машин. Прямая интепретация дерева не очень эффективна (долгий обход по дереву). Поэтому надо эффективно обходить дерево. Классическим примером является язык SECD:

- S - Stack – стек объектов для вычисления выражений
- E - Environment – среда для означивания переменных, контекст
- C - Control – управляющая строка, оставшаяся часть вычисляемого выражения
- D - Dump – стек возвратов для обработки вложенных контекстов (вызовы функций).

Все абстрактные машины являются стековыми, о количестве аргументов заботиться не надо на этапе свертке (очень схоже с алгоритмом прохода-свертки и LR-грамматики).

Данные операции выполняются аналогично eval-apply подходу, но оперируем более простыми выражениями и идем итерационными методами.

Аналогично предыдущим реализациям можно рассмотреть остальные случаи (рекурсия, условия и т.п.).

Рассмотрим редукцию графов. Как устроены правила редукции: ищется самый низкий уровень лямбды, далее смотрим на переменную, а затем меняем ссылки в правой части.

Note 1. The implementation of Functional Programming Languages, 1987

Реализация на потоковых графах: создается взаимодействие функций в виде комбинируемых потоков из схемы функциональных элементов.

Мы не рассмотрели комбинаторную редукцию графов и реализацию на комбинаторной логике.

Как можно реализовать грамматику: происходит лексический анализ, а далее по ним строится синтаксический анализ (к примеру, методом рекурсивного спуска). Но так делать плохо.

Поэтому есть fslex и fsyacc – инструменты для построения всего этого дела, аналогично yacc и lex. В lex строятся токены, а в yacc строится грамматика.

Такой подход не очень удобен (ибо код генерируется плохой).

Remark 5.1. Разве? (Зам. автора)

В F# есть комбинаторный парсер FParsec

```
let str s = pstring s
let float_in_brackets = str “(“ >>. pfloat .>> str “)” // >>. -- return right, .>> -- return left, .>>. --
let floats = sepBy pfloat(str “,”)
...
let pexpr , pexpr_ref = createParserForwarderToRef()
let str’ s = str s .>> ws
let ident = indetifier (new IdentifierOptions()) .>> ws
let pid = ident |>> Var
let plet = pipe (str’ “let”) ident (str ”=”)...
...
do pexpr_ref := choice [pid; plet; pint32 |>> Int; plam; papp; pfunc]
```