Functional / Logic Programming

https://soshnikov.com/courses/funcpro/

Dmitri Soshnikov

http://soshnikov.com

Microsoft / HSE / MIPT / MAI

Functional / Logic Programming

https://soshnikov.com/courses/funcpro/

2. Deeper Dive Into Functional

Dmitri Soshnikov

http://soshnikov.com

Microsoft / HSE / MIPT / MAI

Factorial Revisited

POLYMORPHISM

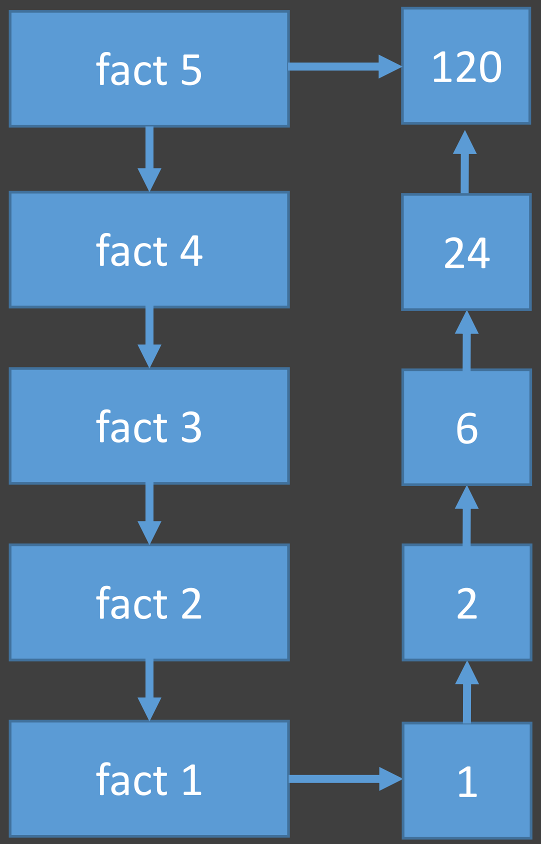
```
let rec iter f i a b =
    if a>b then i
    else f a (iter f i (a+1) b)

let fact = iter (*) 1 1

let bfact = iter (fun i acc -> BigInteger(i)*acc) 1I 1

bfact 12 |> int |> bfact
```

Recursion Problems

```
let rec fact = function
  | x when x=1I -> 1I
  | x -> x*fact(x-1I)
```

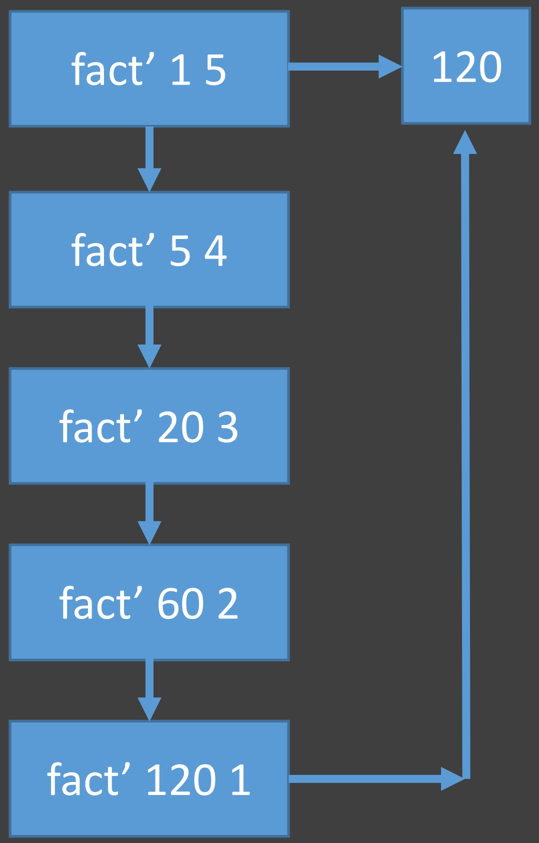
```
13I |> fact |> fact
// causes stack overflow
```

Upon each recursive call, we need to remember return address + all local variables + parameters



Tail Recursion

```
let rec fact' acc = function
  | x when x=1I -> acc
  | x -> fact' (acc*x) (x-1I)

let fact = fact' 1I

8I |> fact |> fact |> printfn "%A"
```



Interesting fact:

$(7!)! =$ [illegible]

How to Write Tail Rec Calls

```
let rec fact = function
  | x when x=1I -> 1I
  | x -> x*fact(x-1I)
```

```
let fact =
 let rec fact' acc = function
      | x when x=1I -> acc
      | x -> fact' (acc*x) (x-1I)
 fact' 1I
```

- Add accumulator as an extra parameter
- Return accumulator upon recursion termination
- Embrace inner function with accumulator with main function

Clever Iter: Be Clever Once

```
let rec iter f i a b =
    if a>b then i
    else iter f (f a i) (a+1) b
```

More on Reduction

- $(\lambda f.\lambda x.f\ 4\ x)\ (\lambda y.\lambda z.+\ z\ y)\ 3$

- $\underline{(\lambda f.\lambda x.f\ 4\ x)\ (\lambda y.\lambda z.+\ z\ y)}\ 3 \rightarrow_{\beta} \underline{(\lambda x.\ (\lambda y.\lambda z.\ +\ z\ y)\ 4\ x)\ 3} \rightarrow_{\beta} \underline{(\lambda y.\lambda z.+\ z\ y)\ 4}\ 3 \rightarrow_{\beta} \underline{(\lambda z.+\ z\ 4)\ 3} \rightarrow_{\beta} +\ 3\ 4$

- Reduction in another order:
- $\underline{(\lambda f.\lambda x.f\ 4\ x)\ (\lambda y.\lambda z.+\ z\ y)}\ 3 \rightarrow_{\beta} (\lambda x.\ \underline{(\lambda y.\lambda z.\ +\ z\ y)\ 4}\ x)\ 3 \rightarrow_{\beta} (\lambda x.\ \underline{(\lambda z.\ +\ z\ 4)\ x})\ 3 \rightarrow_{\beta} \underline{\lambda x.\ (+\ x\ 4)\ 3} \rightarrow_{\beta} +\ 3\ 4$

- Both ways lead to the same result
- Underlined are parts that are being reduced - **redex**

Example

$$\underbrace{(\lambda x.\lambda y.y)\ (\overbrace{(\lambda z.z\ z)\ (\lambda z.z\ z)}^{\text{Inner Redex}})}_{\text{Outer redex}}$$

Importance of the order of reduction

- Consider expression: (λx.λy.y) ((λz.z z) (λz.z z))
- (λx.λy.y) ((λz.z z) (λz.z z)) → λy.y
- (λx.λy.y) ((λz.z z) (λz.z z)) → (λx.λy.y) ((λz.z z) (λz.z z)) → (λx.λy.y) ((λz.z z) (λz.z z)) → …

- First line corresponds to the case when function argument is not computed until it is actually needed. This is called "**call by name**", and corresponds to **lazy computations**.
- In the second case argument is computer first. It is called "**call by value**", or **eager computations**.

Reduction Orders

Applicative Reduction (AR) – leftmost of the inner redexes is reduced

- Eager computations
- More effective in implementation

Normal Reduction (NR) – leftmost of the outer redexes is reduces

- Lazy computations
- Can result in the same expressions computed several times (split problem)
- Difficult to implement some useful side-effects such as IO

Example

- (λx.x+x)(2+3)
  - AR: (λx.x+x) 5 → 5+5 → 10
  - NR: (2+3)+(2+3) → 5+(2+3) →5+5 →10
- In NR same expressions could be computed twice (or more) – split problem
- Several techniques to avoid it:
  - Memoization
  - Computation with context
  - Graph reduction

Standardization theorem

- If expression E has a normal form N, then using NR on each step leads to expression M : $N \approx_\alpha M$
- Proof
- Lazy computations guarantee that all expressions are reducable
- Total (strong) functional programming (more info)
  - Subset of FP, where all functions are defined for all values, and only certain type of recursion is allowed (structural recursion)
  - Not Turing-complete
  - Any expression is computable by any reduction order

Lazy and Eager Languages

- **Eager**
  - F#, ML, OCaml, LISP
  - More efficient
  - Easier to understand (for imperative programmer)
  - Easier to multi-paradigm
- **Lazy**
  - Haskell
  - Pure functional languages
  - More elegant
  - Less efficient

How To Deal With Recursion in Lambda-Calculus

- Recursion can be used when we have naming
- In λ-calculus all functions are anonymous
- Naming construc let is just a syntactic sugar
  - let x=expr1 in expr2 → (λx.expr2)expr1
  - expr1 cannot contain x

Factorial

- let fact x = if x=1 then 1 else x*fact(x-1)
- fact = λx.cond (x=1) 1 (x*fact(x-1))
- fact = (λf.λx.cond (x=1) 1 (x*f(x-1)))fact
- fact = F fact

- fact is called a fixpoint of F
- Consider a fixpoint operator Yf
  - Yf = f(Yf)
- Then fact = YF = Y(λf.λx.cond (x=1) 1 (x*f(x-1)))

Fixpoint operator

- Function can have many, even infinitely many fixpoints
  - $\lambda x.x$
- Yf returns least fixpoint
- $Y = \lambda h.(\lambda x.h(x\ x))\ (\lambda x.h(x\ x))$
- Proof:
  - $Yf = (\lambda h.(\lambda x.h(x\ x))\ (\lambda x.h(x\ x)))\ f \rightarrow (\lambda x.f(x\ x))\ (\lambda x.f(x\ x)) \rightarrow f((\lambda x.f(x\ x))\ (\lambda x.f(x\ x))) = f(Yf)$

Interesting Question!

Can we create a calculus without variables**?**

Combinators

Combinator –λ-expression without free variables

- $Y = \lambda h.(\lambda x.h(x\ x))\ (\lambda x.h(x\ x))$
- $I = \lambda x.x$ – identity function
- $K = \lambda x.\lambda y.x$ – constant function (cancelator)
- $S = \lambda f.\lambda g.\lambda x.fx(gx)$ - distributor

- $I\ x = x$
- $K\ x\ y = x$
- $S\ f\ g\ x = f\ x\ (g\ x)$

Combinatory logic

- Any function can be expressed by using combinators

Basis – a minimal set of combinators that can be used to express functions

  - Example: S,K
  - I = SKK

I = SKK

- SKK = (λf.λg.λx.fx(gx)) (λx.λy.x) (λx.λy.x)
- (λf.λg.λx.fx(gx)) (λu.λv.u) (λs.λt.s)
- (λg.λx. <u>(λu.λv.u)x(gx)</u>) (λs.λt.s)
- (λg.λx.x)(λs.λt.s)
- λx.x = I

Factorial Magic

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

```
let C f g x = f x g
let rec Y f x = f (Y f) x
let cond p f g x = if p x then f x else g x
```

```
let fact = Y (cond ((=)0) (K 1) << (S (*) << ((>>) (C(-)1))))
```

One Trick

```
let fact = Y (fun f -> cond ((=)0) (K 1) (fun n-> n*f(n-1)))
```

$\lambda f.g(h\ f) \equiv g \circ h$ $\lambda x.g(h\ x) \equiv g \circ h$

```
// fun f -> g (h f)  ==> h >> g или g << h
// fun f -> g (E)  ==> g << (fun f -> E)
```

Questions?

- https://soshnikov.com/courses/funcpro/
- http://t.me/funcpro