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

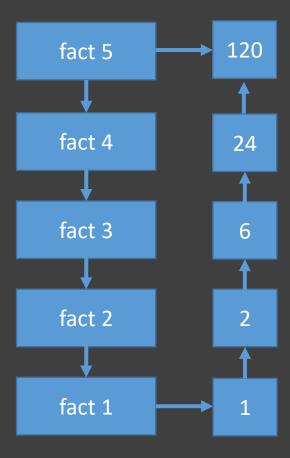
bfact 12 |> int |> bfact
```

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

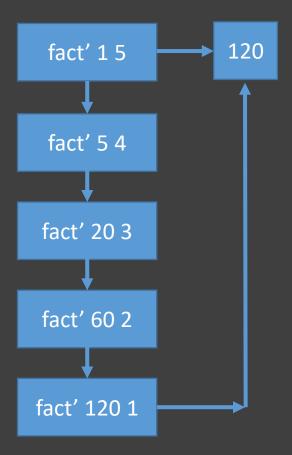


TAIL RECURSION

```
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!)! =

 $60473690202571991119759831856897263046860615540140626105286891475449129429601533808923376531846923113581924171589376867945044745128452960104525036500133911678449633007245113279491092811557863816766263677265381108030639162759389986542822058950029970828950029970828930789757541\\97258529382451553187815154287263613932844108870331450654267059405702903950055742419658015788608130991859726314382117853125949553697852792722187480784022783044080802278075050464770584149035926006774045096043378315457127395481228871426112558849470589091425849550220\\59282960471248120369704347972529312772308744928789620273963957231983139819850918476656449053171949950649434747911416608966868284246610589917175538092306077854812645521262966326085243586798114865849833807580622530285937838312511355396325905485937347469168299736432$ 0.01169481093894153667484984614043453987228494541414116676888155735150423816118062971803476046002629051218507460826997663806769841003390633352867038769666515129839848064390413370065707177105123379350439454404004516677889943626093538580366210348133519796930120

HOW TO WRITE TAIL REC CALLS

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

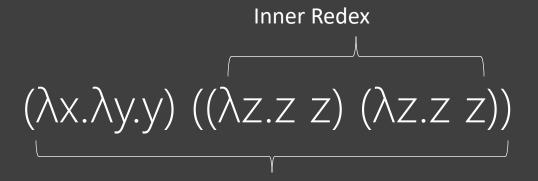
CLEVER ITER: BE CLEVER ONCE

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

MORE ON REDUCTION

- (λf.λx.f 4 x) (λy.λz.+ z y) 3
- $(\lambda f.\lambda x.f + x) (\lambda y.\lambda z. + zy) = 3 \rightarrow_{\beta} (\lambda x. (\lambda y.\lambda z. + zy) + x) = 3 \rightarrow_{\beta} (\lambda y.\lambda z. + zy) = 4 \times 3 \rightarrow_{\beta} (\lambda z. + z + y) = 3 \rightarrow_{\beta} (\lambda y.\lambda z. + zy) = 4 \times 3 \rightarrow_{\beta} (\lambda y.\lambda z. + zy) =$
- Reduction in another order:
- $(\lambda f.\lambda x.f + x) (\lambda y.\lambda z. + zy) = 3 \rightarrow_{\beta} (\lambda x. (\lambda y.\lambda z. + zy) + x) = 3 \rightarrow_{\beta} (\lambda x. (\lambda z. + z + zy) + x) = 3 \rightarrow_{\beta} (\lambda$
- Both ways lead to the same result
- Underlined are parts that are being reduced redex

EXAMPLE



Outer redex

IMPORTANCE OF THE ORDER OF REDUCTION

- Consider expression: (λx.λy.y) ((λz.z z) (λz.z z))
- $(\lambda x.\lambda y.y) ((\lambda z.z z) (\lambda z.z z)) \rightarrow \lambda y.y$
- $(\lambda x.\lambda y.y)$ $(\underline{(\lambda z.z z)}(\lambda z.z z)) \rightarrow (\lambda x.\lambda y.y)$ $(\underline{(\lambda z.z z)}(\lambda z.z z)) \rightarrow (\lambda x.\lambda y.y)$ $(\underline{(\lambda z.z z)}(\lambda z.z z)) \rightarrow ...$
- 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

- $(\lambda x.x + x)(2 + 3)$
 - AR: $(\lambda x.x+x)$ 5 \rightarrow 5+5 \rightarrow 10
 - NR: $(2+3)+(2+3) \rightarrow 5+(2+3) \rightarrow 5+5 \rightarrow 10$
- 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_{α} M
- Proof
- Lazy computations guarantee that all expressions are reducable
- Total (strong) functional programming (<u>more info</u>)
 - 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 $expr2 \rightarrow (\lambda x.expr2)expr1$
 - expr1 cannot contain x

FACTORIAL

- let fact x = if x=1 then 1 else x*fact(x-1)
- fact = $\lambda x.$ cond (x=1) 1 (x*fact(x-1))
- fact = $(\lambda f. \lambda x. cond (x=1) 1 (x*f(x-1))) fact$
- fact = F fact
- fact is called a fixpoint of F
- Consider a fixpoint operator Yf
 - Yf = f(Yf)
- Then fact = YF = $Y(\lambda f.\lambda x.cond(x=1) 1 (x*f(x-1)))$

FIXPOINT OPERATOR

- Function can have many, even infinitely many fixpoints
 - \(\chi_X.X\)
- 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

- $Y = \lambda h.(\lambda x.h(x x)) (\lambda x.h(x x))$
- $I = \lambda x.x identity function$
- $K = \lambda x. \lambda y. x constant function (cancelator)$
- $S = \lambda f. \lambda g. \lambda x. fx(gx) distributor$
- $\bullet \mid \chi = \chi$
- $\bullet K \times y = x$
- S fg x = fx (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
- $\bullet I = SKK$

- SKK = $(\lambda f.\lambda g.\lambda x.fx(gx))(\lambda x.\lambda y.x)(\lambda x.\lambda y.x)$
- (λf.λg.λx.fx(gx)) (λu.λv.u) (λs.λt.s)
- (λg.λx. (λu.λv.u)x(gx)) (λs.λt.s)
- (λg.λx.x)(λs.λt.s)
- $\lambda x.x = 1$

FACTORIAL MAGIC

```
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))))
```

let rec fact n = if n=0 then 1 else n*fact(n-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 \qquad \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