Functional Programming

Specifics

Daniil Berezun

danya.berezun@gmail.com

2022

Operator

- One or more symbols

```
! # $ % & * + . / < > ? @ ^ | - ~ = \ :
```

- all are binary and infix
  except unary minus: `(- x)` ≡ `negate x`.

Operators

```
a *** b = a ^ 2 + b ^ 2
(***) a b = a ^ 2 + b ^ 2
```

Fucntions

```
f a b = a ^ 2 + b ^ 2
a `f` b = a ^ 2 + b ^ 2
```

Priority, sections

Associativity and Priority

```
infixl 9 !!
infixr 9 .
infixr 8 ^, ^^, **
infixl 7 *, /, `quot`, `rem`,
         `div`, `mod`
infixl 6 +, -
infixr 5 ++, :
infix 4 ==, /=, <, <=, >=, >,
        `elem`, `notElem`
infixr 3 &&
infixr 2 ||
infixl 1 >>, >>=
infixr 1 =<<
infixr 0 $, $!, `seq`
```

Function application

- Function application
  highest priority, a-la 10
- $ application

```
infixr 0 $
f $ x = f x
f (g x (h y)) ≡ f $ g x $ h y
```

- Function composition

```
infixr 9 .
f . g = \ x -> f (g x)

f(g(h(e(x)))) ≡
  f . g . h . e $ x
```

Left section

```
(1 *+*) ≡ (*+*) 1 ≡ \x -> 1*+*x
```

Right section

```
(*+* 1) ≡ \x -> x *+* 1
```

LAZY!

Lazy evaluation (call-by-need)

- Delay the evaluation of an expression *until* its *value is needed*
- Also, try to avoid repeated evaluations

Lazy evaluation (call-by-need)

- Delay the evaluation of an expression *until* its *value is needed*
- Also, try to avoid repeated evaluations

Lazy lists

```
ghci> let a = [1..]
ghci> let ones = 1 : ones
ghci> ones
[1,1,1,1,1,1,,1,1,1,Interrupted.
ghci> take 5 ones
[1,1,1,1,1]
```

LAZY!

Lazy evaluation (call-by-need)

- Delay the evaluation of an expression *until* its *value is needed*
- Also, try to avoid repeated evaluations

Lazy lists

```
ghci> let a = [1..]
ghci> let ones = 1 : ones
ghci> ones
[1,1,1,1,1,1,,1,1,1,Interrupted.
ghci> take 5 ones
[1,1,1,1,1]
```

Strict lists

```
data List a = Nil | !a :! !(List a)
ghci> let onesS = 1 :! onesS
ghci> ones
^CInterrupted.
ghci> take 5 ones
^CInterrupted.
```

Lazy evaluation (call-by-need)

- Delay the evaluation of an expression *until* its *value is needed*
- Also, try to avoid repeated evaluations

Lazy lists

```
ghci> let a = [1..]
ghci> let ones = 1 : ones
ghci> ones
[1,1,1,1,1,1,,1,1,1,Interrupted.
ghci> take 5 ones
[1,1,1,1,1]
```

Strict lists

```
data List a = Nil | !a :! !(List a)
ghci> let onesS = 1 :! onesS
ghci> ones
^CInterrupted.
ghci> take 5 ones
^CInterrupted.
```

A step back

How many values of type Bool?

```
data Bool = True | False
```

Lazy evaluation (call-by-need)

- Delay the evaluation of an expression *until* its *value is needed*
- Also, try to avoid repeated evaluations

Lazy lists

```
ghci> let a = [1..]
ghci> let ones = 1 : ones
ghci> ones
[1,1,1,1,1,1,,1,1,1,Interrupted.
ghci> take 5 ones
[1,1,1,1,1]
```

Strict lists

```
data List a = Nil | !a :! !(List a)
ghci> let onesS = 1 :! onesS
ghci> ones
^CInterrupted.
ghci> take 5 ones
^CInterrupted.
```

A step back

How many values of type Bool?

```
data Bool = True | False
```

Hm ... **??? 3 ???** (in some sense)

```
bot :: Bool
bot = not bot
```

LAZY!

Lazy evaluation (call-by-need)

- Delay the evaluation of an expression *until* its *value is needed*
- Also, try to avoid repeated evaluations

Lazy lists

```
ghci> let a = [1..]
ghci> let ones = 1 : ones
ghci> ones
[1,1,1,1,1,1,,1,1,1,Interrupted.
ghci> take 5 ones
[1,1,1,1,1]
```

Strict lists

```
data List a = Nil | !a :! !(List a)
ghci> let onesS = 1 :! onesS
ghci> ones
^CInterrupted.
ghci> take 5 ones
^CInterrupted.
```

A step back

How many values of type Bool?

```
data Bool = True | False
```

Hm ... **??? 3 ???** (in some sense)

```
bot :: Bool
bot = not bot
```

In Haskell's *static* semantics its value is ⊥

```
⊥ :: forall {a} . a
```

Strict?

undefined

```
ghci> undefined
*** Exception: Prelude.undefined
ghci> :t undefined
undefined :: forall {a} . a

f x y = x
ghci> f 42 undefined
42
```

Strict?

undefined

```
ghci> undefined
*** Exception: Prelude.undefined
ghci> :t undefined
undefined :: forall {a} . a

f x y = x
ghci> f 42 undefined
42
```

Force!: seq

```
:info seq
seq :: a -> b -> b
  -- Defined in ‘GHC.Prim’
infixr 0 `seq`
```

We could define seq as follows:

```
seq ⊥ b = ⊥
seq a b = b, if a ≠ ⊥

ghci> seq undefined 42
*** Exception: Prelude.undefined
ghci> seq (id undefined) 42
*** Exception: Prelude.undefined
```

Strict?

undefined

```
ghci> undefined
*** Exception: Prelude.undefined
ghci> :t undefined
undefined :: forall {a} . a

f x y = x
ghci> f 42 undefined
42
```

Force!: seq

```
:info seq
seq :: a -> b -> b
  -- Defined in ‘GHC.Prim’
infixr 0 `seq`
```

We could define seq as follows:

```
seq ⊥ b = ⊥
seq a b = b, if a ≠ ⊥

ghci> seq undefined 42
*** Exception: Prelude.undefined
ghci> seq (id undefined) 42
*** Exception: Prelude.undefined
```

up to weak **WHNF**!

```
ghci> seq (undefined,undefined) 42
42
ghci> seq (\x -> undefined) 42
42
ghci> seq ((+) undefined) 42
42
```

Why to force?

call-by-value

```
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

```
ghci> const 42 $! undefined
*** Exception: Prelude.undefined
```

- Note: $ and $! has the same *type*

```
ghci> :t ($!)
($!) :: (a -> b) -> a -> b
ghci> :t ($)
($) :: (a -> b) -> a -> b
```

Why to force?

call-by-value

```haskell
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

```
ghci> const 42 $! undefined
*** Exception: Prelude.undefined
```

- Note: $ and $! has the same *type*

```
ghci> :t ($!)
($!) :: (a -> b) -> a -> b
ghci> :t ($)
($) :: (a -> b) -> a -> b
```

Do we need to force computations?

```haskell
factorial = helper 1 where
  helper acc k | k > 1 = helper (acc * k) (k - 1)
               | otherwise = acc
```

Why to force?

call-by-value

```
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

```
ghci> const 42 $! undefined
*** Exception: Prelude.undefined
```

- Note: $ and $! has the same *type*

```
ghci> :t ($!)
($!) :: (a -> b) -> a -> b
ghci> :t ($)
($) :: (a -> b) -> a -> b
```

Do we need to force computations?

```
factorial = helper 1 where
  helper acc k | k > 1 = helper (acc * k) (k - 1)
               | otherwise = acc
```

What acc really stores?

Why to force?

call-by-value

```haskell
infixr 0 $!
($!) :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

```
ghci> const 42 $! undefined
*** Exception: Prelude.undefined
```

- Note: $ and $! has the same *type*

```
ghci> :t ($!)
($!) :: (a -> b) -> a -> b
ghci> :t ($)
($) :: (a -> b) -> a -> b
```

Do we need to force computations?

```haskell
factorial = helper 1 where
  helper acc k | k > 1 = helper (acc * k) (k - 1)
               | otherwise = acc
```

What acc really stores? (...((1 * n) * (n - 1)) * (n - 2) * ... * 2)

```haskell
factorial = helper 1 where
  helper acc k | k > 1 = (helper $! acc * k) (k - 1)
               | otherwise = acc
```

Example

```haskell
 -- module Data.Complex
infix 6 :+
data Complex a = !a :+ !a
```

```
ghci> case (1,undefined) of (_,_) -> 42
42
ghci> case 1 :+ undefined of _ :+ _ -> 42
*** Exception: Prelude.undefined
```

```
ghci> case 1 :+ undefined of _ -> 42
42
```

A little more on forcing

Example

```
 -- module Data.Complex
infix 6 :+
data Complex a = !a :+ !a
```

```
ghci> case (1,undefined) of (_,_) -> 42
42
ghci> case 1 :+ undefined of _ :+ _ -> 42
*** Exception: Prelude.undefined
```

```
ghci> case 1 :+ undefined of _ -> 42
42
```

BangPatterns extension

```
ghci> :set -XBangPatterns
ghci> foo !x = True
ghci> foo undefined
*** Exception: Prelude.undefined
```

Lazy and Let...in...

In NAÏVE laziness implementation

Haskell is different

⇓

Repeated computations

```
f x = ((1+2) + x, 4 * x)
```

```
f (5+6)
  → ((1+2)+(5+6), 4*(5+6))
  → (3 + (5+6), 4 * (5+6))
  → (3+11, 4*(5+6))
  → (14, 4*(5+6))
  → (14, 4*11)
  → (14, 44)
```

Let

```
f y = let x = y in ((1+2)+x,4*x)
```

```
f (5+6)
  → let x = (5+6) in ((1+2)+x, 4*x)
  → let x = (5+6) in (3+x, 4*x)
  → let x = 11 in (3+x, 4*x)
  → let x = 11 in (3+11, 4*x)
  → let x = 11 in (14, 4*x)
  → let x = 11 in (14, 4*11)
  → let x = 11 in (14, 44)
  → (11, 44)
```

Lazy patterns

Lazy pattern-matching

```
let (a,b) = p in ...
f ~(a,b) = ...
case p of ~(a,b) -> ...
(\ ~(a,b) -> ... )
```

Example

```
(***)::(a->c)->(b->d)->(a,b)->(c,d)
(***) f g ~(x,y) = (f x, g y)
```

```
ghci> :info const
const :: a -> b -> a
ghci> (const 1 *** const 2) undefined
(1,2)
```

```
f ~(a,b) = g a b
⇒
f p = g (fst p) (snd p)
```

Example

Strict — Lazy

```
h (a,b) = g a b
```

vs

```
f ~(a,b) = g a b
```

```
g _ _ = 42
ghci> f undefined
42
ghci> h undefined
*** Exception: Prelude.undefined
ghci> h (undefined, undefined)
42
```

Lazy patterns: One more example

Example: splitAt

```
splitAt :: Int -> [a] -> ([a], [a])
splitAt n xs =
   if n<=0
   then ([], xs)
   else case xs of
          [] -> ([], [])
          y:ys ->
            case splitAt (n-1) ys of
              ~(prefix, suffix) -> (y : prefix, suffix)
```

```
ghci> take 1000 . fst . splitAt 1000000 $ [1..]
```

- Lazy version produces output **much** fater

No Loops!!

Simpliest HOF

```haskell
twice :: (a -> a) -> a -> a
twice f x = f (f x)
```

why are HOF useful?

- Common programming idioms
- DSLs
- Reasoning about programs

Some useful HOFS: map

```haskell
map :: (a -> b) -> [a] -> [b]
ghci> map (+1) [1,3,5,7]
[2,4,6,8]

map f xs = [f x | x <- xs]

map f []     = []
map f (x:xs) = f x : map f xs
```

Some useful HOFS: filter

```haskell
filter :: (a -> Bool) -> [a] -> [a]

ghci> filter even [1..10]
[2,4,6,8,10]

filter p xs = [x | x <- xs, p x]

filter p [] = []
filter p (x:xs)
  | p x       = x : filter p xs
  | otherwise = filter p xs
```

Common Pattern on Lists

Usual pattern on lists (primitive recursion)

```
f [] = v
f (x:xs) = x ⊕ f xs
```

Examples

```
sum [] = 0
sum (x:xs) = x + sum xs

product [] = 1
product (x:xs) = x * product xs

and [] = True
and (x:xs) =  x && and xs
```

foldr

```
sum = foldr (+) 0
product = foldr (*) 1
or = foldr (||) False
and = foldr (&&) True
```

```
foldr :: (a -> b -> b) -> b ->[a]->b
foldr f v [] = v
foldr f v (x:xs) = f x (foldr f v xs)
```

More on foldr

Thinking foldr non-recursively

- replace (:) with function
- replace value with []

```haskell
sum [1,2,3] = foldr (+) 0 [1,2,3]
            = 1+(2+(3+0))
product [1,2,3]
  = foldr (*) 1 [1,2,3]
  = foldr (*) 1 (1:2:3:[])
  = 1*(2*(3*1))
```

Example: length

```haskell
length :: [a] -> Int
length [] = 0
length (_:xs) = 1 + length xs

length = foldr (\ _ n -> 1+n) 0
```

reverse and ++

```haskell
reverse [] =[]
reverse (x:xs) = reverse xs ++ [x]

reverse =
  foldr (\ x xs -> xs ++ [x]) []
```

```haskell
(++ ys) = foldr (:) ys
```

Summary: foldr

- Simple
- Proving properties
- Advanced optimizations

Declarative!

```
(.) :: (b->c) -> (a->b) -> (a->c)
f . g = \ x -> f (g x)

odd :: Int -> Bool
odd = not . even
```

- **WHAT** to compute instead of **HOW** to compute it

```
sum . map (^2) . filter even
```

Other Useful HOFs on Lists

```haskell
all :: (a -> Bool) -> [a] -> Bool
all p xs = and [p x | x <- xs]

any :: (a -> Bool) -> [a] -> Bool
any p xs = or [p x | x <- xs]

takeWhile :: (a->Bool) ->[a]->[a]
takeWhile p [] = []
takeWhile p (x:xs)
  | p x  = x : takeWhile p ps
  | otherwise = []

dropWhile :: (a->Bool)->[a]->[a]
dropWhile p [] = []
dropWhile p (x:xs)
  | p x       = dropWhile p ps
  | otherwise = x:xs
```

```haskell
foldl

mapAccumL
mapAccumR

zip / unzip
zipWith / unzipWith
```

Some Summary: Haskell

Pros

- ✔ Declarative
- ✔ Compiles $\Rightarrow$ works
- ✔ Program reasoning

Cons

- ✘ Difficult to reason about efficiency
- ✘ Limited tool support for developers
- ✘ Requires ability to think abstractly:
  think then type code

NB: Haskell’s runtime is different
But the reason of stack overflow is the same

Usual recursion

```haskell
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

...	...
main	...
...	...
make 1 3	x=1 n=3 ...

Tail Recursion

NB: Haskell’s runtime is different
But the reason of stack overflow is the same

...	...
main	...
...	...
make 1 3	x=1 n=3 ...
make 1 2	x=1 n=1 ...

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

NB: Haskell's runtime is different
But the reason of stack overflow is the same

...	...
main	...
...	...
make 1 3	x=1 n=3 ...
make 1 2	x=1 n=1 ...
make 1 1	x=1 n=1 ...

Usual recursion

```haskell
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail Recursion

NB: Haskell's runtime is different
But the reason of stack overflow is the same

...	...
main	...
...	...
make 1 3	x=1 n=3 ...
make 1 2	x=1 n=1 ...
make 1 1	x=1 n=1 ...
make 1 0	x=1 n=0 ...

Usual recursion

```haskell
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

NB: Haskell’s runtime is different
But the reason of stack overflow is the same

...	...
main	...
...	...
make 1 3	x=1 n=3 ...
make 1 2	x=1 n=1 ...
make 1 1	x=1 n=1 ...
ret	[]

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail Recursion

NB: Haskell’s runtime is different
But the reason of stack overflow is the same

...	...
main	...
...	...
make 1 3	x=1 n=3 ...
make 1 2	x=1 n=1 ...
ret	[1]

Usual recursion

```haskell
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

NB: Haskell's runtime is different
But the reason of stack overflow is the same

...	...
main	...
...	...
make 1 3	x=1 n=3 ...
ret	[1,1]

Usual recursion

```haskell
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

NB: Haskell’s runtime is different
But the reason of stack overflow is the same

Usual recursion

```haskell
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

...	...
main	...
...	...
ret	[1,1,1]

NB: Haskell's runtime is different
But the reason of stack overflow is the same

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail recursion

```
make2 : a -> Int -> [a]
make2 x n = helper [] n
  helper acc n =
    if n < 1 then acc
    else helper (x F: acc) (n-1)
```

...	...
main	...
...	...
make2 1 3	x=1 n=3 ...

Tail Recursion

NB: Haskell's runtime is different
But the reason of stack overflow is the same

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail recursion

```
make2 : a -> Int -> [a]
make2 x n = helper [] n
  helper acc n =
    if n < 1 then acc
    else helper (x F: acc) (n-1)
```

...	...
main	...
...	...
make2 1 3	x=1 n=3 ...
helper [] 3	acc=[] n=3 ...

NB: Haskell's runtime is different
But the reason of stack overflow is the same

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail recursion

```
make2 : a -> Int -> [a]
make2 x n = helper [] n
  helper acc n =
    if n < 1 then acc
    else helper (x F: acc) (n-1)
```

...	...
main	...
...	...
make2 1 3	x=1 n=3 ...
helper [1] 2	acc=[1] n=2 ...

NB: Haskell's runtime is different
But the reason of stack overflow is the same

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail recursion

```
make2 : a -> Int -> [a]
make2 x n = helper [] n
  helper acc n =
    if n < 1 then acc
    else helper (x F: acc) (n-1)
```

...	...
main	...
...	...
make2 1 3	x=1 n=3 ...
helper [1,1] 1	acc=[1,1] n=1 ...

Tail Recursion

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail recursion

```
make2 : a -> Int -> [a]
make2 x n = helper [] n
  helper acc n =
    if n < 1 then acc
    else helper (x F: acc) (n-1)
```

...	...
main	...
...	...
make2 1 3	x=1 n=3 ...
helper [1,1,1] 0	acc=[1,1,1] n=0 ...

NB: Haskell's runtime is different
But the reason of stack overflow is the same

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail recursion

```
make2 : a -> Int -> [a]
make2 x n = helper [] n
  helper acc n =
    if n < 1 then acc
    else helper (x F: acc) (n-1)
```

...	...
main	...
...	...
make2 1 3	x=1 n=3 ...
ret	[1,1,1]

NB: Haskell’s runtime is different
But the reason of stack overflow is the same

Usual recursion

```
make :: a -> Int -> [a]
make x n = if n < 1 then []
  else x : make x (n-1)
```

Tail recursion

```
make2 : a -> Int -> [a]
make2 x n = helper [] n
  helper acc n =
    if n < 1 then acc
    else helper (x F: acc) (n-1)
```

...	...
main	...
...	...
~~make2 1 3~~	~~x=1 n=3 ...~~
ret	[1,1,1]

Questions?