

Programming Language Concepts

Higher Order Functions

Onur Tolga Şehitoğlu

Computer Engineering



Outline

- 1 Lambda Calculus
- 2 Introduction
- 3 Functions
 - Curry
 - Map
 - Filter
 - Reduce
 - Fold Left
 - Iterate
 - Value Iteration (for)
- 4 Higher Order Functions in C
- 5 Some examples
 - Fibonacci
 - Sorting
 - List Reverse

Lambda Calculus

- 1930's by Alonso Church and Stephen Cole Kleene
- Mathematical foundation for computability and recursion
- Simplest functional paradigm language
- $\lambda var.expr$
defines an anonymous function. Also called **lambda abstraction**
- $expr$ can be any expression with other lambda abstractions and applications. Applications are one at a time.
- $(\lambda x.\lambda y.x + y) 3 4$

- In ' $\lambda var.expr$ ' all free occurrences of var is bound by the λvar .
- Free variables of expression $FV(expr)$
 - $FV(name) = \{name\}$ if $name$ is a variable
 - $FV(\lambda name.expr) = FV(expr) - \{name\}$
 - $FV(M N) = FV(M) \cup FV(N)$
- **α conversion**: expressions with all bound names changed to another name are equivalent:

$$\lambda f.f\ x \equiv_{\alpha} \lambda y.y\ x \equiv_{\alpha} \lambda z.z\ x$$

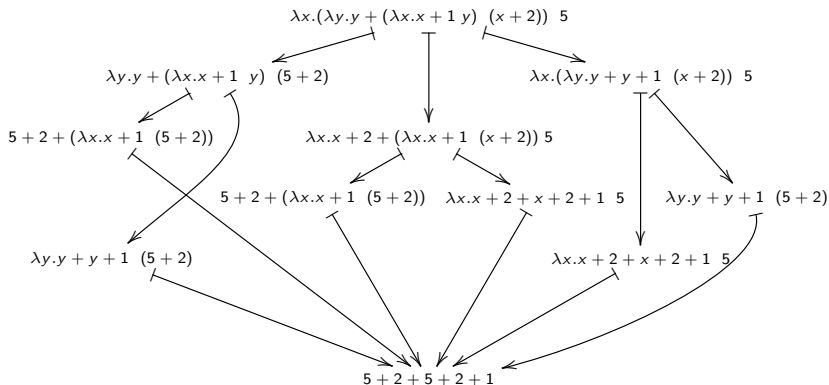
$$\lambda x.x + (\lambda x.x + y) \equiv_{\alpha} \lambda t.t + (\lambda x.x + y) \equiv_{\alpha} \lambda t.t + (\lambda u.u + y)$$

$$\lambda x.x + (\lambda x.x + y) \not\equiv_{\alpha} \lambda x.x + (\lambda x.x + t)$$

β Reduction

- Basic computation step, function application in λ -calculus
- Based on substitution. All bound occurrences of λ variable parameter is substituted by the actual parameter
- $(\lambda x.M)N \mapsto_{\beta} M[x/N]$ (all x 's once bound by lambda are substituted with N).
- $(\lambda x.(\lambda y.y + (\lambda x.x + 1) y)(x + 2)) 5$
- If no further β reduction is possible, it is called a normal form.
- There can be different reduction strategies but should reduce to same normal form. (Church Rosser property)

All possible reductions of a λ -expression. All reduce to the same normal form.



Introduction

■ Mathematics:

$$(f \circ g)(x) = f(g(x)) , (g \circ f)(x) = g(f(x))$$

- “o” : Gets two unary functions and composes a new function.
A function getting two functions and returning a new function.

■ in Haskell:

```
f x = x+x
g x = x*x
compose func1 func2 x = func1 (func2 x)
t = compose f g
u = compose g f
```

- $t \ 3 = (3*3)+(3*3) = 18$

- $u \ 3 = (3+3)*(3+3) = 36$

- $compose: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

Introduction

- Mathematics:

$$(f \circ g)(x) = f(g(x)) , (g \circ f)(x) = g(f(x))$$

- “o” : Gets two unary functions and composes a new function.
A function getting two functions and returning a new function.

- in Haskell:

```
f x = x+x
g x = x*x
compose func1 func2 x = func1 (func2 x)
t = compose f g
u = compose g f
```

- $t \ 3 = (3*3)+(3*3) = 18$

$$u \ 3 = (3+3)*(3+3) = 36$$

- $compose: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

Introduction

- Mathematics:

$$(f \circ g)(x) = f(g(x)) , (g \circ f)(x) = g(f(x))$$

- “o” : Gets two unary functions and composes a new function.
A function getting two functions and returning a new function.

- in Haskell:

```
f x = x+x
g x = x*x
compose func1 func2 x = func1 (func2 x)
t = compose f g
u = compose g f
```

- $t \ 3 = (3*3)+(3*3) = 18$

$$u \ 3 = (3+3)*(3+3) = 36$$

- $\text{compose}: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

Introduction

- Mathematics:

$$(f \circ g)(x) = f(g(x)) , (g \circ f)(x) = g(f(x))$$

- “o” : Gets two unary functions and composes a new function.
A function getting two functions and returning a new function.

- in Haskell:

```
f x = x+x
g x = x*x
compose func1 func2 x = func1 (func2 x)
t = compose f g
u = compose g f
```

- $t \ 3 = (3*3)+(3*3) = 18$

$$u \ 3 = (3+3)*(3+3) = 36$$

- $compose: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

Introduction

- Mathematics:

$$(f \circ g)(x) = f(g(x)) , (g \circ f)(x) = g(f(x))$$

- “o” : Gets two unary functions and composes a new function.
A function getting two functions and returning a new function.

- in Haskell:

```
f x = x+x
g x = x*x
compose func1 func2 x = func1 (func2 x)
t = compose f g
u = compose g f
```

- $t \ 3 = (3*3)+(3*3) = 18$

$$u \ 3 = (3+3)*(3+3) = 36$$

- $\text{compose}: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

- “compose” function is a function getting two functions as parameters and returning a new function.
- Functions getting one or more functions as parameters are called **Higher Order Functions**.
- Many operations on functional languages are repetition of a basic task on data structures.
- Functions are first order values → new general purpose functions that uses other functions are possible.

- “compose” function is a function getting two functions as parameters and returning a new function.
- Functions getting one or more functions as parameters are called **Higher Order Functions**.
- Many operations on functional languages are repetition of a basic task on data structures.
- Functions are first order values → new general purpose functions that uses other functions are possible.

- “compose” function is a function getting two functions as parameters and returning a new function.
- Functions getting one or more functions as parameters are called **Higher Order Functions**.
- Many operations on functional languages are repetition of a basic task on data structures.
- Functions are first order values → new general purpose functions that uses other functions are possible.

- “compose” function is a function getting two functions as parameters and returning a new function.
- Functions getting one or more functions as parameters are called **Higher Order Functions**.
- Many operations on functional languages are repetition of a basic task on data structures.
- Functions are first order values \rightarrow new general purpose functions that uses other functions are possible.

Functions/Curry

- Cartesian form vs curried form:

$$\alpha \times \beta \rightarrow \gamma \text{ vs } \alpha \rightarrow \beta \rightarrow \gamma$$

- Curry function gets a binary function in cartesian form and converts it to curried form.

```
curry f x y = f(x,y)
add (x,y) = x+y
increment = curry add 1
--
increment 5
6
```

- curry: $(\alpha \times \beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \beta \rightarrow \gamma$
- Haskell library includes it as `curry`.

Functions/Map

```

square x = x*x
day no = case no of 1 -> "mon" ; 2 -> "tue" ; 3 -> "wed";
                4 -> "thu" ; 5 -> "fri" ; 6 -> "sat" ; 7 -> "sun"
map func [] = []
map func (el:rest) = (func el):(map func rest)
-----
map square [1,3,4,6]
[1,9,6,36]
map day [1,3,4,6]
["mon","wed","thu","sat"]

```

- $\text{map}:(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$
- Gets a function and a list. Applies the function to all elements and returns a new list of results.
- Haskell library includes it as `map`.

Functions/Filter

```

iseven x = if mod x 2 == 0 then True else False
isgreater x = x>5

filter func [] = []
filter func (el:rest) = if func el then
                        el:(filter func rest)
                        else (filter func rest)

----
filter iseven [1,2,3,4,5,6,7]
[2,4,6]
filter isgreater [1,2,3,4,5,6,7]
[6,7]

```

- $\text{filter}:(\alpha \rightarrow \text{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$
- Gets a boolean function and a list. Returns a list with only members evaluated to True by the boolean function.
- Haskell library includes it as `filter`.

Functions/Reduce (Fold Right)

```

sum x y = x+y
product x y = x*y

reduce func s [] = s
reduce func s (el:rest) = func el (reduce func s rest)
-----
reduce sum 0 [1,2,3,4]
10                                // 1+2+3+4+0
reduce product 1 [1,2,3,4]
24                                // 1*2*3*4*1

```

- $\text{reduce}:(\alpha \rightarrow \beta \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow \beta$
- Gets a binary function, a list and a seed element. Applies function to all elements right to left with a single value.

$$\text{reduce } f \ s \ [a_1, a_2, \dots, a_n] = f \ a_1 \ (f \ a_2 \ (\dots (f \ a_n \ s)))$$
- Haskell library includes it as `foldr`.

- Sum of a numbers in a list:
`listsum = reduce sum 0`
- Product of a numbers in a list:
`listproduct = reduce product 1`
- Sum of squares of a list:
`squaresum x = reduce sum 0 (map square x)`

Functions/Fold Left

```

subtract x y = x - y
foldl func s [] = s
foldl func s (el:rest) =
    foldl func (func s el) rest
----
reduce subtract 0 [1,2,3,4]
-2                                // 1-(2-(3-(4-0)))
foldl subtract 0 [1,2,3,4]
-10                               // (((0-1)-2)-3)-4)

```

- $\text{foldl}:(\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow [\beta] \rightarrow \alpha$
- Reduce operation, left associative.:
 $\text{reduce } f \ s \ [a_1, a_2, \dots, a_n] = f \ (f \ (f \ \dots (f \ s \ a_1) \ a_2 \ \dots)) \ a_n$
- Haskell library includes it as `foldl`.

Functions/Iterate

```
twice x = 2*x

iterate func s 0 = s
iterate func s n = func (iterate func s (n-1))
----
iterate twice 1 4
16                                // twice (twice ( twice (twice 1)))
iterate square 3 3
6561                             // square (square (square 3))
```

- $\text{iterate}:(\alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{int} \rightarrow \alpha$
- Applies same function for given number of times, starting with the initial seed value. $\text{iterate } f \ s \ n = f^n s = \underbrace{f(f(f \dots (f \ s)))}_n$

Functions/Value Iteration (for)

```

for func s m n =
    if m>n then s
    else for func (func s m) (m+1) n

----
for sum 0 1 4
10          // sum (sum (sum (sum 0 1) 2) 3) 4
for product 1 1 4
24          // product (product (product (product 1 1) 2) 3) 4

```

- $\text{for}:(\alpha \rightarrow \text{int} \rightarrow \alpha) \rightarrow \alpha \rightarrow \text{int} \rightarrow \text{int} \rightarrow \alpha$
- Applies a binary integer function to a range of integers in order.

$$\text{for } f \text{ s } m \text{ n} = f(f(f(f(f \text{ s } m) (m+1)) (m+2)) \dots) n$$

- multiply (with summation):
`multiply x = iterate (sum x) x`
- integer power operation (Haskell '^'):
`power x = iterate (product x) x`
- sum of values in range 1 to n:
`seriesum = for sum 0 1`
- Factorial operation:
`factorial = for product 1 1`

Higher Order Functions in C

C allows similar definitions based on function pointers. Example: `bsearch()` and `qsort()` functions in C library.

```
typedef struct Person { char name[30]; int no;} person;
int cmpnmba(void *a, void *b) {
    person *ka=(person *)a; person *kb=(person *)b;
    return ka->no - kb->no;
}
int cmpnames(void *a, void *b) {
    person *ka=(person *)a; person *kb=(person *)b;
    return strcmp(ka->name,kb->name,30);
}
int main() {
    int i;
    person list []={{ "veli",4},{ "ali",12},{ "ayse",8},
                    { "osman",6},{ "fatma",1},{ "mehmet",3}};
    qsort(list,6,sizeof(person),cmpnmba);
    ...
    qsort(list,6,sizeof(person),cmpnames);
    ...
}
```

Fibonacci

Fibonacci series: 1 1 2 3 5 8 13 21 ..

$$fib(0) = 1 ; fib(1) = 1 ; fib(n) = fib(n-1) + fib(n-2)$$

```
fib n = let f (x,y) = (y,x+y)
        (a,b) = iterate f (0,1) n
        in b

----
fib 5      // f(f(f(f(0,1))))
8          //(0,1)->(1,1)->(1,2)->(2,3)->(3,5)->(5,8)
```

Sorting

Quicksort:

- 1** First element of the list is x and rest is xs
- 2** select smaller elements of xs from x , sort them and put before x .
- 3** select greater elements of xs from x , sort them and put after x .

```
notfunc f x y = not (f x y)

sort _ [] = []
sort func (x:xs) = (sort func (filter (func x) xs)) ++
                  (x: (sort func (filter ((notfunc func) x) xs)))
---
sort (>) [5,3,7,8,9,3,2,6,1]
[1,2,3,3,5,6,7,8,9]
sort (<) [5,3,7,8,9,3,2,6,1]
[9,8,7,6,5,3,3,2,1]
```

List Reverse

■ Taking the reverse

- First element is x rest is xs
- Reverse the xs , append x at the end

Loose time for appending x at the end at each step (N times append of size N).

■ Fast version, extra parameter (initially empty list) added:

- Take the first element, insert at the beginning of the extra parameter.
- Recurse rest of the list with the new extra parameter.
- When recursion at the deepest, return the extra parameter.

Inserts to the beginning of the list at each step. Faster (N times insertion)

```
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]

reverse2 x = reverse2 ' x [] where
  reverse2 ' [] x = x
  reverse2 ' (x:xs) y = reverse2 ' xs (x:y)

---
reverse1 [1..10000]      // slow
reverse2 [1..10000]      // fast
```

List Reverse

■ Taking the reverse

- First element is x rest is xs
- Reverse the xs , append x at the end

Loose time for appending x at the end at each step (N times append of size N).

■ Fast version, extra parameter (initially empty list) added:

- Take the first element, insert at the beginning of the extra parameter.
- Recurse rest of the list with the new extra parameter.
- When recursion at the deepest, return the extra parameter.

Inserts to the beginning of the list at each step. Faster (N times insertion)

```
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]

reverse2 x = reverse2 ' x [] where
  reverse2 ' [] x = x
  reverse2 ' (x:xs) y = reverse2 ' xs (x:y)

---
reverse1 [1..10000] // slow
reverse2 [1..10000] // fast
```

List Reverse

■ Taking the reverse

- First element is x rest is xs
- Reverse the xs , append x at the end

Loose time for appending x at the end at each step (N times append of size N).

■ Fast version, extra parameter (initially empty list) added:

- Take the first element, insert at the beginning of the extra parameter.
- Recurse rest of the list with the new extra parameter.
- When recursion at the deepest, return the extra parameter.

Inserts to the beginning of the list at each step. Faster (N times insertion)

```
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]

reverse2 x = reverse2 ' x [] where
  reverse2 ' [] x = x
  reverse2 ' (x:xs) y = reverse2 ' xs (x:y)

--
reverse1 [1..10000] // slow
reverse2 [1..10000] // fast
```

List Reverse

■ Taking the reverse

- First element is x rest is xs
- Reverse the xs , append x at the end

Loose time for appending x at the end at each step (N times append of size N).

■ Fast version, extra parameter (initially empty list) added:

- Take the first element, insert at the beginning of the extra parameter.
- Recurse rest of the list with the new extra parameter.
- When recursion at the deepest, return the extra parameter.

Inserts to the beginning of the list at each step. Faster (N times insertion)

```
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]

reverse2 x = reverse2 ' x [] where
    reverse2 ' [] x = x
    reverse2 ' (x:xs) y = reverse2 ' xs (x:y)

--
reverse1 [1..10000] // slow
reverse2 [1..10000] // fast
```

List Reverse

■ Taking the reverse

- First element is x rest is xs
- Reverse the xs , append x at the end

Loose time for appending x at the end at each step (N times append of size N).

■ Fast version, extra parameter (initially empty list) added:

- Take the first element, insert at the beginning of the extra parameter.
- Recurse rest of the list with the new extra parameter.
- When recursion at the deepest, return the extra parameter.

Inserts to the beginning of the list at each step. Faster (N times insertion)

```
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]

reverse2 x = reverse2 ' x [] where
  reverse2 ' [] x = x
  reverse2 ' (x:xs) y = reverse2 ' xs (x:y)

--
reverse1 [1..10000] // slow
reverse2 [1..10000] // fast
```


List Reverse

■ Taking the reverse

- First element is x rest is xs
- Reverse the xs , append x at the end

Loose time for appending x at the end at each step (N times append of size N).

■ Fast version, extra parameter (initially empty list) added:

- Take the first element, insert at the beginning of the extra parameter.
- Recurse rest of the list with the new extra parameter.
- When recursion at the deepest, return the extra parameter.

Inserts to the beginning of the list at each step. Faster (N times insertion)

```
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]

reverse2 x = reverse2 ' x [] where
    reverse2 ' [] x = x
    reverse2 ' (x:xs) y = reverse2 ' xs (x:y)

--
reverse1 [1..10000] // slow
reverse2 [1..10000] // fast
```

List Reverse

■ Taking the reverse

- First element is x rest is xs
- Reverse the xs , append x at the end

Loose time for appending x at the end at each step (N times append of size N).

■ Fast version, extra parameter (initially empty list) added:

- Take the first element, insert at the beginning of the extra parameter.
- Recurse rest of the list with the new extra parameter.
- When recursion at the deepest, return the extra parameter.

Inserts to the beginning of the list at each step. Faster (N times insertion)

```
reverse1 [] = []
reverse1 (x:xs) = (reverse1 xs) ++ [x]

reverse2 x = reverse2 ' x [] where
    reverse2 ' [] x = x
    reverse2 ' (x:xs) y = reverse2 ' xs (x:y)

--
reverse1 [1..10000]      // slow
reverse2 [1..10000]      // fast
```