Functional Programming
Lecture 7

Rostislav Horčík

Czech Technical University in Prague
Faculty of Electrical Engineering
xhorcik@fel.cvut.cz

Lambda calculus

Acknowledgement

Lecture based on

Raúl Rojas : *A Tutorial Introduction to the Lambda Calculus*, FU Berlin , WS 97/98. `https://arxiv.org/abs/1503.09060`

Link is also provided in CourseWare.

Untyped lambda calculus

A formalism introduce by Alonzo Church in 1930s.

The simplest universal programming language

- function definition scheme (λ-abstraction)
- variable substitution rule (α-conversion, β-reduction)

Introduced as a tool to prove that not all functions are computable.

λ-calculus is Turing-complete.

It serves as a formal basis for functional programming languages.

Syntax

Syntax

A program in λ-calculus is an expression

```
<expr> -> <var> | <function> | <application>
<function> -> (λ<var>.<expr>)
<application> -> (<expr> <expr>)
```

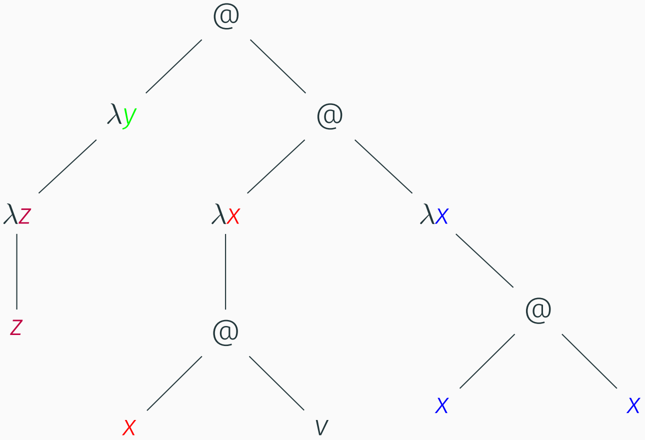
Conventions:

- We often leave the outermost parentheses.
- The application is left-associative, e.g. $e_1 e_2 e_3 e_4$ is $(((e_1 e_2) e_3) e_4)$
- The bodies of functions extends to the right as far as possible.

$$\lambda x.(\lambda y.xyx)z \equiv \left(\lambda x.\left((\lambda y.((xy)x))z\right)\right)$$

Abstract Syntax Tree

$$(\lambda y.(\lambda z.z))\Big((\lambda x.(xv))(\lambda x.(xx))\Big)$$



Free and bound variables

A variable in an λ-expression is **bound** if it is under the scope of λ and **free** otherwise.

Bound variable names can be renamed anytime by a fresh variable, e.g.

$$\lambda x.xz \equiv \lambda y.yz$$

The renaming process is called **α-conversion**.

An expression is **closed** (aka **combinator**) if it has no free variables; otherwise it is **open**.

Semantics

β-reduction

Lambda term $(\lambda x.t)$ represents a function with an argument x and body t.

In Racket `(lambda (x) t)`

Function can by applied to another expression:

$$((\lambda x.e_1)e_2) \qquad \textbf{redex}$$

It is applied by substituting the free occurrences of x in e_1 by e_2.

$$((\lambda x.e_1)e_2) \to^{\beta} e_1[x := e_2]$$

Examples of β-reductions

$$(\lambda x.x)(\lambda y.y) \to^{\beta} x[x := (\lambda y.y)] \equiv (\lambda y.y)$$

$$\begin{aligned}(\lambda x.xx)(\lambda y.y) &\to^{\beta} (xx)[x := (\lambda y.y)] \\ &\equiv (\lambda y.y)(\lambda y.y) \to^{\beta} (\lambda y.y)\end{aligned}$$

$$(\lambda x.x(\lambda x.x))y \to^{\beta} (x(\lambda x.x))[x := y] \equiv y(\lambda x.x)$$

Name conflicts

Avoid name conflicts by renaming bound variables (α-conversion)

1. Do not let a substituent become bound

$$(\lambda x.(\lambda y.xy))y \not\rightarrow^{\beta} (\lambda y.yy)$$
$$(\lambda x.(\lambda y.xy))y \equiv (\lambda x.(\lambda z.xz))y \rightarrow^{\beta} (\lambda z.yz)$$

2. Substitute only the free occurrences of argument

$$(\lambda x.(\lambda y.x(\lambda x.xy)))z \not\rightarrow^{\beta} (\lambda y.z(\lambda x.zy))$$
$$(\lambda x.(\lambda y.x(\lambda x.xy)))z \rightarrow^{\beta} (\lambda y.z(\lambda x.xy))$$

Evaluation strategies

Expression may contain several redexes $\underbrace{(\lambda x.x)(\overbrace{(\lambda y.y)z}^{\text{redex}})}_{\text{redex}}$

- **Normal order**: reduce **leftmost outermost** redex first
- **Applicative order**: reduce **leftmost innermost** redex first

Expression with no redex is in **normal form**.

Reduction process need not terminate!

$$(\lambda x.xx)(\lambda x.xx) \to^{\beta} (\lambda x.xx)(\lambda x.xx)$$

Church-Rosser Theorems

1. Normal forms are unique (independently of eval. strategy).
2. Normal order always finds a normal form if it exists.

Leftmost outermost vs leftmost innermost redex

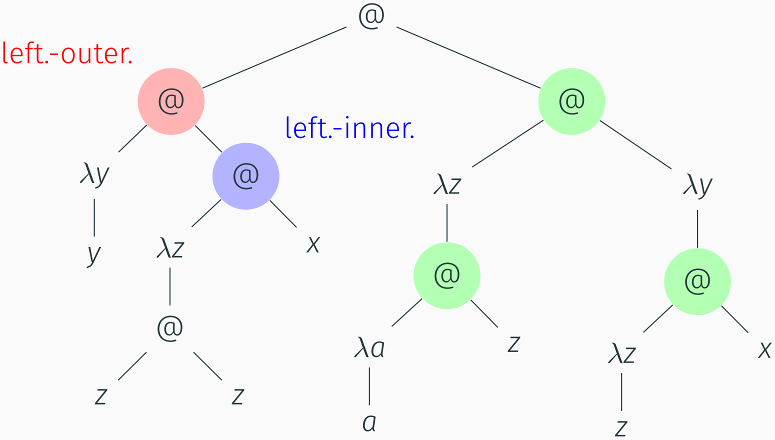
We say a redex is to the left of another redex if its lambda appears further left.

The leftmost outermost redex is the leftmost redex not contained in any other redex.

The leftmost innermost redex is the leftmost redex not containing any other redex.

Leftmost outermost vs leftmost innermost redex

$$(\lambda y.y)((\lambda z.zz)x)\Big((\lambda z.(\lambda a.a)z)(\lambda y.(\lambda z.z)x)\Big)$$



Building Booleans, arithmetic, recursion, etc. in λ-calculus

No function names

Functions in λ-calculus do not have names.

We apply a function by writing its whole definition.

We use capital letters and symbols to abbreviate the definitions. These abbreviations are not a part of λ-calculus.

E.g. the identity function is usually abbreviated by I

$$I \equiv (\lambda x.x)$$

Combinators are the building blocks.

Boolean values

Lambda term of the form $\lambda x.(\lambda y.e)$ is is abbreviated $\lambda xy.e$.

$$T \equiv \lambda xy.x$$
$$F \equiv \lambda xy.y$$

The T and F functions directly serve as the if-statement

$$Tab \to^{\beta} a$$
$$Fab \to^{\beta} b$$

Logical operations

Conjunction:

$$\land \equiv \lambda xy.xyF \equiv \Big(\lambda x\Big(\lambda y.((xy)(\lambda uv.v))\Big)\Big)$$

Disjunction:

$$\lor \equiv \lambda xy.xTy \equiv \lambda xy.x(\lambda uv.u)y$$

Negation:

$$\neg \equiv \lambda x.xFT \equiv \lambda x.x(\lambda uv.v)(\lambda ab.a)$$

$$\land FT \equiv (\lambda xy.xyF)FT \rightarrow^{\beta} FTF \rightarrow^{\beta} F$$

$$\land TT \equiv (\lambda xy.xyF)TT \rightarrow^{\beta} TTF \rightarrow^{\beta} T$$

Natural numbers

Natural numbers are represented as functions of two variables s, z so that n is represented as n-fold application of s to z.

$$
\begin{aligned}
0 &\equiv \lambda sz.z \equiv F \\
1 &\equiv \lambda sz.sz \\
2 &\equiv \lambda sz.s(sz) \\
3 &\equiv \lambda sz.s(s(sz)) \\
&\vdots
\end{aligned}
$$

Successor function

Increment a number by one

$$S \equiv \lambda \textcolor{red}{w} yx.y(\textcolor{red}{w} yx)$$

E.g.

$$\begin{aligned} S1 &\equiv (\lambda wyx.y(wyx))(\lambda sz.sz) \\ &\rightarrow^{\beta} \lambda yx.y((\lambda sz.sz)yx) \\ &\rightarrow^{\beta} \lambda yx.y(yx) \equiv 2 \end{aligned}$$

Addition

$x + y$ is applying the successor x times to y

Meaning of number N is just ”apply the first argument N times to the second argument”

$$N \equiv \lambda sz.\, \underbrace{s(s \ldots (s}_{N \text{ times}}\, z) \ldots)$$

Therefore $2 + 3$ is just:

$$2S3 \equiv (\lambda sz.s(sz))S3 \rightarrow^{\beta} S(S3) \rightarrow^{\beta} 5$$

Multiplication

We can multiply two numbers using

$$M \equiv \lambda abc.a(bc)$$

Note

$$Nc \equiv (\lambda sz.\, \underbrace{s(s \ldots (s}_{N \text{ times}}\, z) \ldots))c \rightarrow^{\beta} \lambda z.\, \underbrace{c(c \ldots (c}_{N \text{ times}}\, z) \ldots)$$

$$\begin{aligned} M23 &\equiv (\lambda abc.a(bc))23 \rightarrow^{\beta} \lambda c.2(3c) \\ &\rightarrow^{\beta} \lambda c.(\lambda z.(3c)((3c)z)) \equiv \lambda cz.(3c)((3c)z) \\ &\rightarrow^{\beta} \lambda cz.(3c)(c(c(cz))) \rightarrow^{\beta} \lambda cz.c(c(c(c(c(cz))))) \equiv 6 \end{aligned}$$

Conditional tests

Test if a given number is the 0

$$Z \equiv \lambda x.xF\neg F$$

$$Z0 \equiv (\lambda x.xF\neg F)0 \to^{\beta} 0F\neg F \to^{\beta} \neg F \to^{\beta} T$$

For $N > 0$

$$\begin{aligned} ZN &\equiv (\lambda x.xF\neg F)N \to^{\beta} NF\neg F \\ &\to^{\beta} (\underbrace{F \ldots (F}_{N \text{ times}} \neg) \ldots)F \to^{\beta} IF \to^{\beta} F \end{aligned}$$

because

$$Fe \equiv (\lambda ab.b)e \to^{\beta} \lambda b.b \equiv I$$

Pairs

The pair $\langle a, b \rangle$ can be represented as

$$\langle a, b \rangle \equiv \lambda z.zab$$

We can extract the first element of the pair by

$$(\lambda z.zab)T \to^{\beta} a$$

and the second element by

$$(\lambda z.zab)F \to^{\beta} b$$

Predecessor

We want to create a function, which applied N times to something returns $N-1$.

This function modifies a pair $\langle x, y\rangle$ to $\langle x+1, x\rangle$

$$\Phi \equiv \lambda pz.z(S(pT))(pT)$$

Calling N times Φ on $\langle 0, 0\rangle$ yields $\langle N, N-1\rangle$.

$$\Phi\langle 0, 0\rangle \to^{\beta} \langle 1, 0\rangle, \quad \Phi\langle 1, 0\rangle \to^{\beta} \langle 2, 1\rangle, \quad \ldots$$

Finally, we take the second number in the pair. The predecessor function is

$$P \equiv \lambda n.n\Phi\langle 0, 0\rangle F$$

Note than the predecessor of 0 is 0.

Y-combinator

Can we create recursion without function names?

$$Y \equiv \lambda y.(\lambda x.y(xx))(\lambda x.y(xx))$$

Now apply Y to some other function R

$$\begin{aligned} YR &\rightarrow^{\beta} (\lambda x.R(xx))(\lambda x.R(xx)) \equiv \tilde{R} \\ &\rightarrow^{\beta} R((\lambda x.R(xx))(\lambda x.R(xx))) \equiv R\tilde{R} \\ &\rightarrow^{\beta} R(R((\lambda x.R(xx))(\lambda x.R(xx)))) \equiv R(R\tilde{R}) \\ &\rightarrow^{\beta} \dots \end{aligned}$$

$$\tilde{R} \rightarrow^{\beta} R\tilde{R}$$

Recursive functions

We can recursively sum up first *n* integers as

$$\sum_{i=0}^{n} i = n + \sum_{i=0}^{n-1} i$$

In Racket

```
(define (sum-to n)
  (if (= n 0) 0
    (+ n (sum-to (- n 1)))))
```

A corresponding recursive function is

$$R \equiv \lambda rn.Zn0(nS(r(Pn)))$$

Recursive functions

$$R \equiv \lambda rn.Zn0(nS(r(Pn)))$$

$$\begin{aligned}
YR3 &\to^{\beta} \tilde{R}3 \to^{\beta} R\tilde{R}3 \equiv Z30(3S(\tilde{R}(P3))) \\
&\to^{\beta} F0(3S(\tilde{R}(P3))) \to^{\beta} 3S(\tilde{R}(P3)) \\
&\to^{\beta} 3S(\tilde{R}2) \to^{\beta} 3S(R\tilde{R}2) \\
&\to^{\beta} 3S(Z20(2S(\tilde{R}(P2)))) \to^{\beta} 3S(2S(\tilde{R}1)) \\
&\to^{\beta} 3S(2S(R\tilde{R}1)) \to^{\beta} 3S(2S(1S(\tilde{R}0))) \\
&\to^{\beta} 3S(2S(1S(R\tilde{R}0))) \equiv 3S(2S(1S(Z00(0S(\tilde{R}(P0)))))) \\
&\to^{\beta} 3S(2S(1S0)) \to^{\beta} 6
\end{aligned}$$

What have we learned?

- λ-calculus is the formal basis for functional programming language.
- It is the simplest universal programming language.
- It uses only λ-abstraction and application.
- Within λ-calculus it is possible to build up numbers, arithmetic, etc.
- Recursion is done via Y-combinator.