Functional Programming

Lecture 7: Lambda calculus

Viliam Lisý
Rostislav Horčík
Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague
viliam.lisy@fel.cvut.cz
xhorcik@fel.cvut.cz

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.

Lambda calculus

Theory developed for studying properties of effectively computable functions

Formal basis for functional programming

- as Turing machines for imperative programming

Smallest universal programming language

- function definition scheme
- variable substitution rule

Introduced by Alonzo Church in 1930s

Why to care?

- Understand that lambda and application is enough to build any program
  - without mutable state, assignment, define, etc.
  - useful for proving program properties
- Understand how numbers, conditions, recursion can be created in a purely functional way
- Think about programming yet a little differently
- Have a clue when someone mentions λ-calculus
- Understand that scheme syntax is not the worst

Syntax

A program in λ-calculus is an expression

<expression> := <name> | <function> | <application>

<function> := λ <name>.<expression>

<application> := <expression><expression>

Names, also called a variables, will be *x,y,z,…*

By convention

$E_1E_2E_3 \ldots E_n$ is interpreted as $(\ldots(E_1E_2)E_3)\ldots E_n)$

Free and bound variables

A variable in a body of a function is **bound** if it is under the scope of λ and **free** otherwise.

$\lambda x.\, x\boldsymbol{y}, (\lambda x.\, x)(\lambda y.\, y\boldsymbol{x})$ - bold variables are free

Bound variable names can be renamed anytime

$$\lambda x.\, x \equiv \lambda y.\, y \equiv \lambda z.\, z$$

An expression is **closed** if it has no free variables; otherwise it is **open**.

β-reduction

Lambda term represents a function:

Argument Body

$$\lambda x.\, e_1$$

In scheme `(lambda (x) e1)`

Function can by applied to an expression e_2

$$(\lambda x.\, e_1)e_2$$ redex

It is applied by substituting free x's in e_1 by e_2

$$(\lambda x.\, e_1)e_2 = [e_2/x]e_1$$

Examples of β-reduction

- $(\lambda x.\ x)\ (\lambda y.\ y) \equiv [(\lambda y.\ y)/x]\ x \equiv (\lambda y.\ y)$
- $(\lambda x.\ x\ x)\ (\lambda y.\ y) \equiv [(\lambda y.\ y)/x](x\ x)$

  $\equiv (\lambda y.\ y)\ (\lambda y.\ y)$
- $(\lambda x.\ x\ (\lambda x.\ x))\ y \equiv [y/x](x\ (\lambda x.\ x))$

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

Name conflicts

Avoid name conflicts by renaming bound variables

1) do not let a substituent become bound

$$\big(\lambda x.(\lambda y.xy)\big)y \text{ does } \textbf{not} \text{ yield } \lambda y.yy$$

$$[y/x](\lambda z.xz) = \lambda z.yz$$

2) substitute only free occurrences of argument

$$\Big(\lambda x.\Big(\lambda y.\big(x(\lambda x.xy)\big)\Big)\Big)z \text{ is } \textbf{not} \ (\lambda y.(z(\lambda z.zy)))$$

$$[z/x](\lambda y.(x(\lambda x.xy))) = (\lambda y.(z(\lambda x.xy)))$$

Reduction orders

Expression may contain several redexes.

- Normal-order reduction: reduce left-most first
- Applicative-order reduction: reduce right-most first

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

Reduction process need not terminate!

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

Church-Rosser Theorem:

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

Non-naming of functions

Function in λ-calculus do not have names

- We apply a function by writing its whole definition
- We use capital letters and symbols to abbreviate this
  - These function names are not a part of λ-calculus

The identity function is usually abbreviated by I

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

Conditionals

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

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

The T and F functions directly serve as If

$$Tab = a$$
$$Fab = b$$

Logical operations

AND

$$\wedge \equiv \lambda xy.xy(\lambda uv.v) \equiv \lambda xy.xyF$$

OR

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

Negation

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

$$\wedge TF \equiv (\lambda xy.xyF)TF \equiv TFF \equiv (\lambda ab.a)FF \equiv F$$

$$\wedge FT \equiv (\lambda xy.xyF)FT \equiv FTF \equiv (\lambda ab.b)TF \equiv F$$

Numbers

We define a "zero" and a successor function representing the next number

$$0 \equiv \lambda s.(\lambda z.z) \equiv \lambda sz.z$$

$$1 \equiv \lambda sz.s(z)$$

$$2 \equiv \lambda sz.s\big(s(z)\big)$$

$$3 \equiv \lambda sz.s(s(s(z)))$$

Functional alternative of binary representation

Successor function

Increment a number by one

$$S \equiv \lambda wyx.\, y(wyx)$$

Increment zero to get one

$$S0 \equiv \big(\lambda wyx.\, y(wyx)\big)(\lambda sz.\, z) =$$
$$\lambda yx.\, y\big((\lambda sz.\, z)yx\big) =$$
$$\lambda yx.\, y\big((\lambda z.\, z)x\big) =$$
$$\lambda yx.\, y(x) \equiv 1$$

Try: $S1$, S2,…

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"

$$\lambda sz.\, s(\dots s(s(z)) \dots)$$

Therefore 2+3 is just:

$$2S3 \equiv$$
$$(\lambda sz.\, s(sz))(\lambda wyx.\, y(wyx))(\lambda uv.\, u(u(uv)))$$
$$\equiv S(S3) \equiv S4 \equiv 5$$

Multiplication

We can multiply two numbers using

$$* \equiv (\lambda xyz.x(yz))$$

$$* \, 2 3 \equiv \big(\lambda xyz.x(yz)\big)2 3 = \big(\lambda z.2(3z)\big) =$$

$$\Big(\lambda z.(\lambda xy.x\big(x(y)\big))(3z)\Big) =$$

$$\Big(\lambda z.(\lambda y.(3z)\big((3z)(y)\big))\Big) =$$

$$\left(\lambda z.(\lambda y.\left(z\left(z\left(z\big((3z)(y)\big)\right)\right)\right))\right) =$$

$$\left(\lambda zy.\left(z\left(z\left(z(z(z\big(z(y)\big)))\right)\right)\right)\right) = 6$$

Conditional tests

Test if a given number is the 0

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

$$Z0 \equiv$$
$$(\lambda x. xF\neg F)0 = 0F\neg F = \neg F = T$$

$$ZN \equiv$$
$$(\lambda x. xF\neg F)N = NF\neg F$$
$$= F(\ldots F(\neg) \ldots)F = IF = F$$

Pairs

The pair $[a, b]$ can be represented as

$$(\lambda z. zab)$$

We can extract the first element of the pair by

$$(\lambda z. zab)T$$

and the second element by

$$(\lambda z. zab)F$$

Predecessor

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

The function modifies a pair (x, y) to $(x+1, x)$

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

Calling Φ on $[0,0]$ N times yields $[N, N-1]$

$$\Phi[0,0] = [1,0] \qquad \Phi[1,0] = [2,1] \quad \ldots$$

Finally, we take the second number in the pair

The predecessor function is

$$P \;\equiv\; \lambda n.n\Phi(\lambda z.z00)F$$

Note than the predecessor of 0 is 0

Equality and inequality

$x \geq y$ can be represented by

$$G \equiv (\lambda xy.Z(xPy))$$

Equality if than defined based on the above as

$$E \equiv \lambda xy.\wedge(Gxy)(Gyx)$$
$$\equiv (\lambda xy.\wedge(Z(xPy))(Z(yPx)))$$

Other inequalities can be defined analogically using the previously defined logical operations

Recursion

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

$$YR \equiv \big(\lambda x.R(xx)\big)\big(\lambda x.R(xx)\big) \equiv$$

$$R((\lambda x.R(xx))(\lambda x.R(xx))) \equiv$$

$$R(YR) \equiv R(R(YR)) \equiv R(\ldots(R(YR)\ldots)$$

Function R is called with YR as the first argument

Recursion

We can recursively sum up first n integers as

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

In scheme

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

A corresponding recursive function is

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

Recursion

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

$$YR3 \equiv$$
$$R(YR)3 \equiv Z30\left(3S\left(YR(P3)\right)\right) \equiv$$
$$F0\left(3S\left(YR(P3)\right)\right) \equiv \left(3S\left(YR(P3)\right)\right) \equiv$$
$$3S(YR2) \equiv 3S\left(2S(YR1)\right) \equiv 3S2S1S0 \equiv 6$$

Turing completeness

Turing machine

- a standard formal model of computation
- B4B01JAG Jazyky, automaty a gramatiky
- what can be solved by TM, can be solved by standard computers

A programming language Turing complete, if it can solve all problems solvable by TM

Lambda calculus is Turing complete

Summary

- Lambda calculus is formal bases of FP
- Simplest universal programming language
- Everything using lambda and application
  - conditions
  - numbers
  - pairs
  - recursion