Functional Programming

Lecture 7: Lambda calculus

Viliam Lisý

Artificial Intelligence Center
Department of Computer Science
FEE, Czech Technical University in Prague

viliam.lisy@fel.cvut.cz

Acknowledgement

Lecture based on:

Raúl Rojas: *A Tutorial Introduction to the Lambda Calculus,* FU Berlin, WS-97/98.

Link will be 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 do I care?

- Understand that lambda and application is enough to build any program
  - without mutable state, assignment, define, etc.
- 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>
| (<expression>)

<function> := λ <name>.<expression>

<application> := <expression><expression>

Names, also called a variables, will be a,b,c,…

By convention

$E_1 E_2 E_3 \ldots E_n$ is interpreted as $(\ldots(E_1 E_2) E_3) \ldots E_n)$

Function application

Identity function

Argument Body

$$\lambda x.\, x$$

Function can by applied to expression

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

Function is applied by substituting arguments

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

Free and bound variables

A variable in a body of a function is **bound** if it is an argument of the function 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

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

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

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

Example

$$II \equiv$$
$$(\lambda x.\, x)(\lambda y.\, y)$$
$$[\lambda y.\, y/x\,]x = \lambda y.\, y \equiv I$$

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

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

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

Conditionals

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

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

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

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

$$\lambda xy.\, x(\ldots x(x(y)) \ldots)$$

Meaning of number n is just "apply the first argument n times to the second argument"

Therefore 2+3 is just:

$$2S3 \equiv$$
$$(\lambda sz.\, s(sz))(\lambda wyx.\, y(wyx))(\lambda uv.\, u(u(uv)))$$
$$= SS3 = S4 = 5$$

Multiplication

We can multiply two numbers using

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

$$* 23 \equiv \left(\lambda xyz.x(yz)\right)23 = \left(\lambda z.2(3z)\right) =$$
$$\left(\lambda z.(\lambda xy.x\left(x(y)\right))(3z)\right) =$$
$$\left(\lambda z.(\lambda y.(3z)\left((3z)(y)\right))\right) =$$
$$\left(\lambda z.(\lambda y.\left(z\left(z\left(z\left((3z)(y)\right)\right)\right)\right))\right) =$$
$$\left(\lambda zy.\left(z\left(z\left(z(z(z\left(z(y)\right)))\right)\right)\right)\right) = 6$$

Conditional tests

Test if a given number is the 0

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

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

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

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 $[N, N-1]$

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

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

$$E \equiv \lambda xy.\wedge GxyGyx = (\lambda xy.\wedge (Z(xPy))(Z(yPx)))$$

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

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

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

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

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