Lecture 2

Lambda calculus

Iztok Savnik, FAMNIT

February, 2021.

Literature

Henk Barendregt, Erik Barendsen, Introduction to Lambda Calculus, March 2000.

Lambda calculus

- Leibniz had as ideal the following

  1) Create a 'universal language' in which all possible problems can be stated.

  2) Find a decision method to solve all the problems stated in the universal language.

- (1) was fulfilled by

  – Set theory + predicate calculus (Frege,Russel,Zermelo)

- (2) has become important philosophical problem:

  – Can one solve all problems formulated in the universal language?

  – Entscheidungsproblem

Entscheidungsproblem

- Negative outcome
- Alonzo Church, 1936
  - Proposes LC as extension of logic
  - Shows the existance of undecidable problem
  - Functional programming languages
- Alan Turing, 1936
  - Proposes TM
  - Turing proved that both models define the same class of computable functions
  - Corresponds to Von Neumann computers
  - Imperative programming languages

Functions

- Function is basic concept of classical and modern mathematics
- Let *A* and *B* be sets and let *f* be relation.
  - *dom(f) = X*
  - $\forall x \in A$: $\exists$ unique $y \in B$ such that *(x,y)* $\in f$
  - Uniquness: *(x,y)* $\in f \wedge$ *(x,z)* $\in f \Rightarrow y=z$
  - *f* maps or transforms *x* to *y*
- *f : A → B*
  - *f* is function from *A to B*

Lambda notation

- **Lambda expression**
  - Pure lambda calculus expression includes
    - variables: *x, y, z, ...*
    - lambda abstraction: $\lambda x.M$
    - application*:   M N*
- **Lambda abstraction** $\lambda x.M$ *represents* **function**
  - *x* is function argument
  - *M* is function expression
    - Receipt that specifies how function is »computed«
- **Application** *M N*
  - If $M = \lambda x.M'$ then all occurences of *x* in *M'* are replaced with *N*
  - Mechanical definition of parameter passing

On notations

- Let *x + 1* be expression with variable *x*
  - Mathematical notation: *f(x) = x + 1*
  - Lambda notation: λx.(x + 1)
- Let *x + y* be expression where *x* and *y* are variables
  - Mathematical notation: *f(x,y) = x + y*
  - Lambda notation: *λx.λy.(x + y)*
- Obvious difference:
  - *λ*-notation does not name function

Definition of LC syntax

Definition: The set of λ-expressions Λ is constructed from infinite set of variables *{v,v',v'',v''', ...}* by using *application* and *λ-abstraction*:

$$x \in V \Rightarrow x \in \Lambda$$

$$M, N \in \Lambda \Rightarrow (M\ N) \in \Lambda$$

$$X \in V,\ M \in \Lambda \Rightarrow \lambda x.M \in \Lambda$$

Backus-Naur form of λ-calculus syntax:

M ::= V | (λv.M) | (M N)

V ::= v, v' ,v'' . . .

Syntax rules

- *Application* is left-associative

$$M \ N \ L \equiv (M \ N) \ L$$

- *Λ-abstraction* is right-associative

$$\lambda x.\lambda y.\lambda z.M \ N \ L \equiv \lambda x.(\lambda y.(\lambda z.((M \ N) \ L)))$$

- We often use the following abbreviation

$$\lambda xyz.M \equiv \lambda x.\lambda y.\lambda z.M$$

Examples

- Let's see some examples of *λ*-expression

$$y$$
$$yx$$
$$λx.y\ x$$
$$(λx.y\ x)\ z$$
$$(λx.λy.y\ x)\ z\ w$$
$$(λu.λf.λx.f\ (u\ f\ x))\ (λv.λy.v\ y)$$

Free and bound variables

- Abstraction *λx.M* binds variable *x* in expression *M*
  - In simmilar manner the function argumens are bound to the function body
- *M* is scope of variable *x* in experssion *λx.M*
- *Variable x is free* in some expression *M* if there exist no *λ*-abstraction that binds it
- Name of free variable is important while the name of bound variable is not
- Example:

$$λx.(x + y)$$

Computing free variables

Definition: The set of free variables of λ-expression *M,* denoted *FV(M),* is defined with the following rules:

$$FV(x) = \{x\}$$
$$FV(M\ N\ ) = FV(M) \cup FV(N)$$
$$FV(\lambda x.M) = FV(M) - \{x\}$$

Example:

$$FV(\lambda x.x\ (\lambda y.x\ y\ z)) = \{z\}$$

Definition: λ-expression *M* is closed if *FV(M)={}*.

Substitution

- Substitution is the basis of LC evaluation
  - Computing is string rewriting ?
- Substitute all instances of a variable *x* in *λ*-expression *M* with *N*:

$$[N/x]M$$

Definition: Let M,N∈Λ and x,z∈V. Substitution rules:

[N/x]x = N

[N/x]z = z, if z ≠ x

[N/x](L M) = ([N/x]L)([N/x]M)

[N/x](λz.M) = λz.([N/x]M), if z≠x ∧ z∉FV(N)

Example

- *[y(λv.v)/x]λz.(λu.u) z x ≡ λz.(λu.u) z (y (λv.v))*
- Check evaluation of substitution rules !

Alpha conversion

- Renaming bound variables in λ-expression yields equivalent λ-expression
- Example:

$$\lambda x.x \equiv \lambda y.y$$

- *Alpha conversion* rule:

$$\lambda x.M \equiv \lambda y.([y/x]M), \text{ if } y \notin FV(M).$$

Example

- Λ-expression:

$$(\lambda f.\lambda x.f\ (f\ x))\ (\lambda y.y + x)$$

- Blind substitution gives:

$$\lambda x.((\lambda y.y + x)\ ((\lambda y.y + x)x)) = \lambda x.x + x + x$$

- Correct substitution:

$$\lambda z.((\lambda y.y + x)\ ((\lambda y.y + x)\ z)) = \lambda z.z + x + x$$

Evaluation

- Λ-calculus is very expressive language equivalent to Turing machine
- Evaluation of λ-expressions is based on:
  1) α-coversion and

  2) substitution
- Evaluation is often called reduction
- Λ-expressions are reduced to value
  - Values are normal forms of λ-expressions i.e. λ-expressions that can not be further reduced

β-reduction

- β-reduction is the only rule used for evaluation of pure λ-calculus (aside from renaming)

- Expression *(λx.M) N* stands for operator *(λx.M)* applied to parameter *N*

- Intuitive interpretation of *(λx.M) N* is substitution of *x* in *M* for *N*

β-reduction

Definition: Let *λx.M* be λ-expression. Application of *(λx.M)* on parameter *N* is implemented with β-reduction:

$$(λx.M) \; N \rightarrow [N/x]M$$

- Expression (λx.M) N is called redex (reducable expression)
- Expression [N/x]M is called contractum

β-reduction

- *P* includes redex *(λx.M) N that is substituted with [N/x]M* and we obtain *P'*

- We say that *P* *β-reduces* to *P':*

$$P \rightarrow_\beta P'$$

Definition: *β-derivation* is composed of one or more *β-reductions.* *β-derivation from M to N:*

$$M \twoheadrightarrow_\beta N$$

Simple examples

- $(\lambda x.x\ y)(u\ v) \rightarrow_\beta u\ v\ y$

- $(\lambda x.\lambda y.x)z\ w \rightarrow_\beta (\lambda y.z)w \rightarrow_\beta z$

  $(\lambda x.\lambda y.x)z\ w \twoheadrightarrow_\beta z$

- $(\lambda x.(\lambda y.yx)z)v \rightarrow [v/x](\lambda y.yx)z = (\lambda y.yv)z$
  $$\rightarrow [z/y]yv = zv$$

More examples

- Example with identity function
  $(\lambda x.x)E \rightarrow [E/x]x = E$

- Another example with identity function
  $(\lambda f.f (\lambda x.x))(\lambda x.x) \rightarrow$
  $[(\lambda x.x)/f]f (\lambda x.x) = [(\lambda x.x)/f]f (\lambda y.y) \rightarrow$
  $(\lambda x.x)(\lambda y.y) \rightarrow$
  $[(\lambda y.y)/x]x = \lambda y.y$

Tricky examples

- Repeating *β-derivation*

  (λx.xx)(λy.yy)

  → [(λy.yy)/x]xx = (λx.xx)(λy.yy)

  → [(λy.yy)/x]xx = (λx.xx)(λy.yy)

  → ...

- Counting *β-derivation:*

  (λx.xxy)(λx.xxy)

  → [(λx.xxy)/x]xxy = (λx.xxy)(λx.xxy)y

  → ([(λx.xxy)/x]xxy)y = (λx.xxy)(λx.xxy)yy → …

Higher-order functions

- Higher-order function is a function that can either:
  - take another function as an argument, or,
  - return function as the result of function application.
- Example:
  - Construct compositum: $(f \circ f)(x) = f(f(x))$
  - Lambda expression: $\lambda f.\lambda x.f\ (f\ x)$

$$(\lambda f.\lambda x.f\ (f\ x))(\lambda y.y + 1)$$
$$= \lambda x.(\lambda y.y + 1)((\lambda y.y + 1)x)$$
$$= \lambda x.(\lambda y.y + 1)(x + 1)$$
$$= \lambda x.(x + 1) + 1$$

Higher-order functions

- The same function (f∘f)(x) in Lisp

  (lambda(f)(lambda(x)(f (f x))))

  ((lambda(f)(lambda(x)(f (f x))))(lambda(y)(+ y 1))
  = (lambda(x)((lambda(y)(+ y 1))((lambda(y)(+ y 1)) x))))
  = (lambda(x)((lambda(y)(+ y 1))(+ x 1))))
  = (lambda(x)(+ (+ x 1) 1))

Programming in LC

- Function in Curry form
- Combinators
  - Primitives of programming languages
- Logical values
  - If statement
- Integer numbers
  - Arithmetics
- Recursion

Curry functions

- Functions can have single parameter in λ-calculus
- Multiple parameters can be implemented by using higher-order functions
- F is function with parameters (N, L) and body M
  - M be expression with free variables x and y
  - We wish to replace x with N and y with L
- Curry notation: F ≡ λx.λy.M
  - F N L → (λy.[N/x]M)L → [L/y][N/x]M
  - Λ-calculus with pairs: F ≡ λ(x,y).M
- Transformation from λ(x,y).M to λx.λy.M is called Currying

Combinators

- Combinators are primitive functions
  - Expressing basic operations of computation
  - Functions: identity, composition, choice, etc.
- Combinatory logic CL
  - Curry, Feys, 1958
  - Combinators are building blocks of CL
  - CL uses combinators **I**, **K** and **S**
- Combinators are often used in programming languages
  - Higher-order functions: apply, map, fold, filter, etc.

Combinators

- Identity function:

$$\mathbf{I} = \lambda x.x$$

- Choosing one argument of two:

$$\mathbf{K} = \lambda x.(\lambda y.x)$$

- Passing argument to two functions:

$$\mathbf{S} = \lambda x.\lambda y.\lambda z.(x\ z)(y\ z)$$

- Function that repeats itself:

$$\mathbf{\Omega} = (\lambda x.x\ x)(\lambda x.x\ x)$$

- Function composition:

$$\mathbf{B} = \lambda f.\lambda g.\lambda x.f(g\ x)$$

Combinators

- Inverse function composition:

$$\mathbf{B'} = \lambda f.\lambda g.\lambda x.g(f\ x)$$

- Duplication of function argument:

$$\mathbf{W} = \lambda f.\lambda x.f\ x\ x$$

- Recursive function:

$$\mathbf{Y} = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$$

Logical values

- How to represent truth (logical) values?
  - true ≡ λt.λf.t | function returning first argument of two
  - false ≡ λt.λf.f | function returning second argument of two
- IF statement is simple application of truth value
  - λl.λm.λn. l m n
  - Truth value determines first or second choice
- Evaluation of IF statement

  IF true M N ≡ (λl.λm.λn. l m n) true M N →

  (λm.λn. true m n) M N →

  true M N = (λt.λf.t) M N →

  (λf.M) N → M

Church numbers

$$C_0 = \lambda z.\lambda s.z$$
$$C_1 = \lambda z.\lambda s.s\ z$$
$$C_2 = \lambda z.\lambda s.s(s\ z)$$
$$\ldots$$
$$C_n = \lambda z.\lambda s.s(s(\ldots(s\ z)\ldots))$$

- Number n is represented with C_n
  - n = 0+1+...+1 | n times successor of 0
  - z stands for zero and s represents successor function
- Arithmetic operations
  - Plus = $\lambda m.\lambda n.\lambda z.\lambda s.m\ (n\ z\ s)\ s$
  - Times = $\lambda m.\lambda n.m\ C_0\ (\text{Plus } n)$

Church numbers

(Plus 1 2) →* 3

Plus (λz.λs.s z) (λz.λs.s(s z)) →
(λm.λn.λz.λs.m(n z s)s) (λz.λs.s z) (λz.λs.s(s z)) →
(λn.λz.λs.(λz.λs.s z)(n z s)s) (λz.λs.s(s z)) →
λz.λs.(λz.λs.s z)((λz.λs.s(s z)) z s)s →
λz.λs.(λz.λs.s z)((λs.s(s z)) s)s →
λz.λs.(λz.λs.s z)(s(s z))s =
λz.λs.(((λz.λs.s z) (s(s z)))s) →
λz.λs.((λs.s(s (s z)))s) →
λz.λs.s(s (s z))

Recursion

- Recursion can be expressed using combinator **Y**
  - Y = λf.(λx.f (x x))(λx.f (x x))
- Important property of **Y**
  - Y F =$_β$ F (Y F)
  - Proof:

    Y F = λf.(λx.f (x x))(λx.f (x x)) F →
        (λx.F (x x))(λx.F (x x)) →
        F ((λx.F (x x))(λx.F (x x))) ←
        F ((λf.(λx.f (x x))(λx.f (x x))) F) =
        F (Y F)

Recursion

if n = 0 then 1
else n ∗ (if n − 1 = 0 then 1
else (n − 1) ∗ (if n − 2 = 0 then 1
else (n − 2) ∗ . . .

- Operation factorial: n!
  - Intuitive definition
- Definition of recursive function F
  - G = λf.M | M is body of f
  - F = Y G
- Derivation of F

F = Y G

$=_β$ G (Y G)

$=_β$ M (Y G)

$=_β$ M (M (Y G))

...

Factorial

Fact = λfact.λn.if (IsZero n) C1 (Times n (fact(Pred n)))

Factorial = Y Fact

Factorial C2 = Y Fact C2

$=_\beta$ Fact (Y Fact) C2

$=_\beta$ (λfact.λn.if (IsZero n) C1 (Times n(fact(Pred n)))) (Y Fact) C2

$=_\beta$ (λn.if (IsZero n) C1 (Times n(Y Fact (Pred n)))) C2

$=_\beta$ if (IsZero C2) C1 (Times C2 (Y Fact (Pred C2)))

$=_\beta$ if False C1 (Times C2 (Y Fact C1)))

$=_\beta$ Times C2 (Y Fact C1)

= Times C2 (Factorial C1)

β-normal form

Definition: 1) λ-expression *Q* that does not include *β-redexes* is in *β-normal form.*

2) The class of all *β-normal forms* is called *β-nf.*

3) If *P* β-reduces to *Q,* which is β-nf, then *Q* is β-normal form of *P.*

Examples

- Let L ≡ (λx.xxy)(λx.xxy).

$$L \to Ly \to Lyy \to ...$$

- Let P ≡ (λu.v)L. P can be reduced in two ways.
  - P ≡ (λu.v)L → ([L/u]v)L ≡ v
  - P → (λu.v)Ly

    → (λu.v)Lyy

    → ...

 P has β-nf but also infinite derivation !

Evaluation order

- Some λ-expressions can be reduced in more than one way.

- Example:

    1) (λx.(λy.yx)z)v ➞ (λy.yv)z ➞ zv

    2) (λx.(λy.yx)z)v ➞ (λx.zx)v ➞ zv

- Evaluation strategies:
    - Normal form strategie (left-outer redex first)
    - Call by name (no reductions in λ-abstractions + nf)
    - Call by value (outer redex but after right-hand side reduced)

Church-Rosser theorem

Theorem: Let P $\twoheadrightarrow_\beta$ M and P $\twoheadrightarrow_\beta$ N, then there exists T such that M $\twoheadrightarrow_\beta$ T and N $\twoheadrightarrow_\beta$ T.
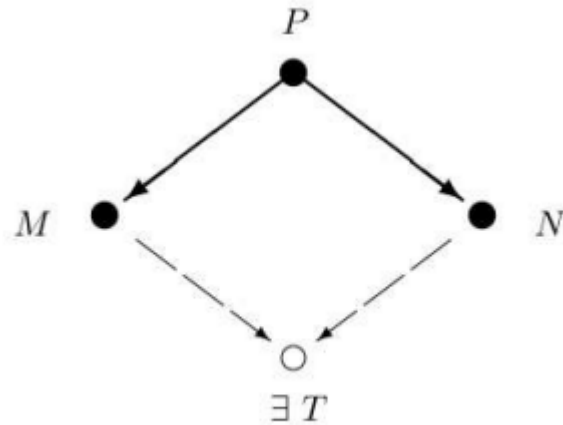
Consequences of CR

- $M =_\beta N \Rightarrow \exists L: M \twoheadrightarrow_\beta L \land N \twoheadrightarrow_\beta L$
- If N is β-nf of expression M then $M \twoheadrightarrow_\beta N$
- Every expression has exactly one β-nf
- Consistency of λ-calculus: $\Lambda \nvdash true =_\beta false$

Some properties of LC

- LC is consistent
- LC is equivalent to TM (Turing machine)
  - LC is r.e.
  - LC is partially computable (not total !)
- LC with types is total function
  - Very limited class of languages
- The characterisation of total TM is not known