Lecture 2

Lambda calculus

Iztok Savnik, FAMNIT

March, 2022.

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 $(x,y) \in f$
  - Uniqness: $(x,y) \in f \wedge (x,z) \in f \Rightarrow y=z$
  - *f* maps or transforms *x* to *y*
- $f : A \to B$
  - *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: $\lambda x.(x + 1)$
- Let $x + y$ be expression where x and y are variables
  - Mathematical notation: $f(x,y) = x + y$
  - Lambda notation: $\lambda x.\lambda y.(x + y)$
- 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:

$$M ::= V \mid (\lambda v.M\ ) \mid (M\ N)$$

$$V ::= v,\ v'\ , v''\ .\ .\ .$$

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
  - Notice spaces!

$$y$$

$$y\ x$$

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

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

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

Examples: λ and ocaml

Some λ-expressions (notice spaces!):

```
3
λx.x
(λx.x) (λy.y ∗ y)
(λz.z + 1) 3
```

In OCaml:

```
# 3;;
- : int = 3
# function x -> x;;
- : ’a -> ’a = <fun>
# (function x -> x) (function y->y*y);;
- : int -> int = <fun>
# (function z -> z + 1) 3;;
- : int = 4
```

Free and bound variables

- Abstraction $\lambda x.M$ 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 $\lambda x.M$
- *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:

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

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:

$$[N/x]x = N$$

$$[N/x]z = z, \text{ if } z \neq x$$

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

$$[N/x](\lambda z.M\ ) = \lambda z.([N/x]M\ ), \text{ if } z \neq x\ \wedge\ z \notin FV(N)$$

Example

- $[y(\lambda v.v)/x]\lambda z.(\lambda u.u)\ z\ x \ \equiv\ \lambda z.(\lambda u.u)\ z\ (y\ (\lambda v.v))$
  - 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:

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

Example: Alpha conversion

- 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 $(\lambda x.M)\ N$ stands for operator $(\lambda x.M)$ applied to parameter N
- Intuitive interpretation of $(\lambda x.M)\ N$ 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:

$$(\lambda x.M)\ N \rightarrow [N/x]M$$

- 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:*

$$M \twoheadrightarrow_{\beta} N$$

Examples of the evaluation

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

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

  $\rightarrow [z/y]yv = zv$

Examples of the evaluation

- Example with identity function

  $(\lambda x.x)E \to [E/x]x = E$

- Another example with identity function

  $(\lambda f.f\ (\lambda x.x))(\lambda x.x) \to$

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

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

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

Examples of the evaluation

- Repeating *β-derivation*

  $(\lambda x.xx)(\lambda y.yy)$

  $\rightarrow [(\lambda y.yy)/x]xx = (\lambda x.xx)(\lambda y.yy)$

  $\rightarrow [(\lambda y.yy)/x]xx = (\lambda x.xx)(\lambda y.yy)$

  $\rightarrow$ ...

- Counting *β-derivation:*

  $(\lambda x.xxy)(\lambda x.xxy)$

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

  $\rightarrow ([(\lambda x.xxy)/x]xxy)y = (\lambda x.xxy)(\lambda x.xxy)yy \rightarrow \ldots$

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: $\lambda f.\lambda x.f\ (f\ x)$

$$
\begin{aligned}
& (\lambda f.\lambda x.f\ (f\ x))(\lambda y.y + 1) \\
= {} & \lambda x.(\lambda y.y + 1)((\lambda y.y + 1)\ x) \\
= {} & \lambda x.(\lambda y.y + 1)(x + 1) \\
= {} & \lambda x.(x + 1) + 1
\end{aligned}
$$

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))
```

Examples in Ocaml

```
# let c = 4;;
val c : int = 4
# let sq = function x -> x*x;;      (* λx.x*x *)
val sq : int -> int = <fun>
# let nx = function x -> x + 1;;    (* λx.x+1 *)
val nx : int -> int = <fun>
#
# let compose = function f -> function g -> function x -> f(g(x));;    (* λf.λg.λx.f(g(x)) *)
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let rcompose = function f -> function g -> function x -> g(f(x));;   (* λf.λg.λx.g(f(x)) *)
val rcompose : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
# let c1 = compose sq nx;;                  (* c1 = (λf.λg.λx.f(g(x))) (λx.x*x) (λx.x+1) *)
val c1 : int -> int = <fun>
# let c2 = rcompose sq nx;;                 (* c2 = (λf.λg.λx.g(f(x))) (λx.x*x) (λx.x+1) *)
val c2 : int -> int = <fun>
#
# c1 3;;
- : int = 16
# c2 3;;
- : int = 10
```

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: $F \equiv \lambda x.\lambda y.M$
  - $F\ N\ L \rightarrow (\lambda y.[N/x]M\ )L \rightarrow [L/y][N/x]M$
  - Λ-calculus with pairs: $F \equiv \lambda(x,y).M$
- Transformation from $\lambda(x,y).M$ to $\lambda x.\lambda y.M$ is called Currying

Example: Curry functions

- Math notation: sum $\equiv \lambda\langle x,y\rangle.x + y$
  - sum : $\mathbb{Z}\rightarrow\mathbb{Z}\rightarrow\mathbb{Z}$ (type of sum)
- Curry notation: sum $\equiv \lambda x.\lambda y.x + y$
  - Application to the first argument returns a function. funkcijo.
  - suma $\equiv (\lambda x.\lambda y.x + y)\ a \rightarrow \lambda y.a + y$
  - suma : $\mathbb{Z}\rightarrow\mathbb{Z}$
- Ocaml libraries are written in Curry notation
  - New functions can be defined from existing functions.
  - Examples will be presented on the lecture on Functional languages

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
  - Functions that construct new functions (genericity?)
    - Examples will be given when we present fun. languages
  - Higher-order functions: apply, map, fold, filter, etc.

Combinators

- Identity function:

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

- Choosing one argument of two (if):

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

- Passing argument to two functions:

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

- Function that repeats itself (loop):

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

- 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 ≡ $\lambda t.\lambda f.t$ | function returning first argument of two
  - false ≡ $\lambda t.\lambda f.f$ | function returning second argument of two
- IF statement is simple application of truth value
  - $\lambda l.\lambda m.\lambda n.\ l\ m\ n$
  - Truth value determines first or second choice
- Evaluation of IF statement

$$
\begin{aligned}
\text{IF true M N} \equiv\ & (\lambda l.\lambda m.\lambda n.\ l\ m\ n)\ \text{true}\ M\ N \rightarrow \\
& (\lambda m.\lambda n.\ \text{true}\ m\ n)\ M\ N \rightarrow \\
& \text{true}\ M\ N = (\lambda t.\lambda f.t)\ M\ N \rightarrow \\
& (\lambda f.M\ )\ N \rightarrow M
\end{aligned}
$$

Church numbers

$C_0 = \lambda z.\lambda s.z$

$C_1 = \lambda z.\lambda s.s\ z$

$C_2 = \lambda z.\lambda s.s(s\ z)$

...

$C_n = \lambda z.\lambda s.s(s(. . . (s\ z) . . .)$

- 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

$(\text{Plus } 1\ 2) \to^{*}\ 3$

$\text{Plus } (\lambda z.\lambda s.s\ z)\ (\lambda z.\lambda s.s(s\ z)) \to$

$(\lambda m.\lambda n.\lambda z.\lambda s.m(n\ z\ s)s)\ (\lambda z.\lambda s.s\ z)\ (\lambda z.\lambda s.s(s\ z)) \to$

$(\lambda n.\lambda z.\lambda s.(\lambda z.\lambda s.s\ z)(n\ z\ s)s)\ (\lambda z.\lambda s.s(s\ z)) \to$

$\lambda z.\lambda s.(\lambda z.\lambda s.s\ z)((\lambda z.\lambda s.s(s\ z))\ z\ s)s \to$

$\lambda z.\lambda s.(\lambda z.\lambda s.s\ z)((\lambda s.s(s\ z))\ s)s \to$

$\lambda z.\lambda s.(\lambda z.\lambda s.s\ z)(s(s\ z))s =$

$\lambda z.\lambda s.(((\lambda z.\lambda s.s\ z)\ (s(s\ z)))s) \to$

$\lambda z.\lambda s.((\lambda s.s(s\ (s\ z)))s) \to$

$\lambda z.\lambda s.s(s\ (s\ z))$

Recursion

- Recursion can be expressed using combinator **Y**
  - $Y = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$
- Important property of **Y**
  - $Y\ F =_\beta F\ (Y\ F\ )$
  - Proof:

$$
\begin{aligned}
Y\ F = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))\ F &\rightarrow \\
(\lambda x.F\ (x\ x))(\lambda x.F\ (x\ x)) &\rightarrow \\
F\ ((\lambda x.F\ (x\ x))(\lambda x.F\ (x\ x))) &\leftarrow \\
F\ ((\lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x)))\ F\ ) &= \\
F\ (Y\ F\ ) &
\end{aligned}
$$

Recursion

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

$$
\begin{aligned}
&\text{if } n = 0 \text{ then } 1\\
&\text{else } n * (\text{if } n - 1 = 0 \text{ then } 1\\
&\text{else } (n - 1) * (\text{if } n - 2 = 0 \text{ then } 1\\
&\text{else } (n - 2) * \ldots
\end{aligned}
$$

$$
\begin{aligned}
F &= Y\ G\\
&=_\beta G\ (Y\ G)\\
&=_\beta G\ (Y\ G)\\
&=_\beta G\ (G\ (Y\ G))\\
&\ldots
\end{aligned}
$$

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.*

Is every λ-expression normalizable?

- Definitely not!
- Let $L \equiv (\lambda x.xxy)(\lambda x.xxy)$.

$$L \to Ly \to Lyy \to \ldots$$

- Let $P \equiv (\lambda u.v)L$. P can be reduced in two ways.
  - $P \equiv (\lambda u.v)L \to ([L/u]v)L \equiv v$
  - $P \to (\lambda u.v)Ly$

    $\to (\lambda u.v)Lyy$

    $\to \ldots$
- P has β-nf but also infinite derivation!

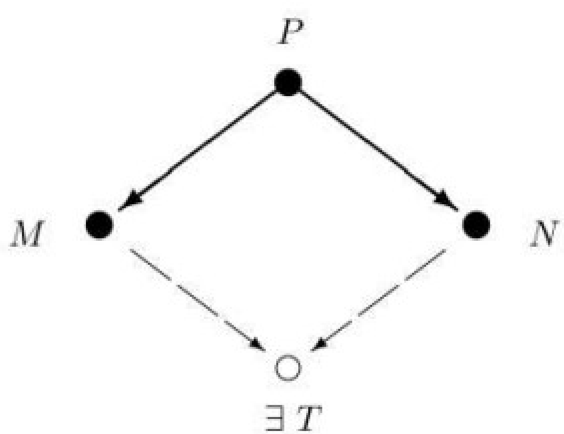
On evaluation order

- Some λ-expressions can be reduced in more than one way.
- Example:
  1) (λx.(λy.y x) z) v → (λy.y v) z → z v
  2) (λx.(λy.y x) z) v → (λx.z x) v → z v
- 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

A central theorem in lambda calculus.

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

- $M =_\beta N \Rightarrow \exists L$: $M \twoheadrightarrow_\beta L \land N \twoheadrightarrow_\beta L$
  - M derivation of (derived from) N ⇒ they have the same value
- If N is β-nf of expression M then $M \twoheadrightarrow_\beta N$
  - N is value of M ⇒ there must be a derivation
- Every expression has exactly one β-nf
  - Consistency of λ-calculus: $\Lambda \nvdash \text{true} =_\beta \text{false}$

Properties of LC

- LC is consistent
- LC is equivalent to TM (Turing machine)
  - LC is r.e. language
  - 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