



# Functional Programming

## Lecture 5

Cezary Kaliszyk   Jonas Schöpf  
Christian Sternagel   Vincent van Oostrom

Department of Computer Science

## Topics

abstract data types, algebraic data types, binary search trees, combinator parsing, efficiency, **encoding data types as lambda-terms**, evaluation strategies, formal verification, first steps, guarded recursion, Haskell introduction, higher-order functions, historical overview, induction, infinite data structures, input and output, **lambda-calculus**, lazy evaluation, list comprehensions, lists, modules, pattern matching, polymorphism, property-based testing, reasoning about functional programs, recursive functions, sets, strings, tail recursion, trees, tupling, type checking, type inference, types, types and type classes, unification, user-defined types

## Overview

- Introduction to the  $\lambda$ -Calculus
- Encoding Data Types

## Origin

- search for general framework in which every algorithm can be defined
- “universal language” (for computation)
- 1936 – Alonzo Church introduces  **$\lambda$ -calculus** in *An Undecidable Problem of Elementary Number Theory*, *AJM* **58**(2), pages 345–363, [doi:10.2307/2371045](https://doi.org/10.2307/2371045)
- 1937 – Alan Turing introduces Turing Machines in *On Computable Numbers with an Application to the Entscheidungsproblem*, *LMS*, **42**(2), pages 230–265, [doi:10.1112/plms/s2-42.1.230](https://doi.org/10.1112/plms/s2-42.1.230)
- and shows both models of computation equivalent
- that is, **Turing-complete** is same as **definable in  $\lambda$ -calculus**
- $\lambda$ -calculus closer to functional programming

Syntax –  $\lambda$ -Terms

- grammar

$$t ::= x \quad \text{variable/atom}$$

$$| (t t) \quad \text{application}$$

$$| (\lambda x. t) \quad \text{(lambda) abstraction}$$

- occurrences of  $x$  in  $\lambda x. t$  are **bound** (by **binder** “ $\lambda x.$ ”)

## Examples

$$(\lambda x. y)$$

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

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

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

## Conventions (to ease reading and writing)

- outermost parentheses are dropped
- application binds stronger than abstraction, e.g.,  $\lambda x. x z$  is equal to  $\lambda x. (x z)$  and **not** to  $(\lambda x. x) z$
- application associates to the left
- nested abstractions associate to the right and variables are combined

## Examples (using Conventions)

$$\lambda x. y$$

$$\lambda x y. x$$

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

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

## Note

- think of nested lambdas as “functions with multiple arguments”
- e.g.,  $\lambda x y z. t$  is function taking 3 arguments

Haskell vs.  $\lambda$ -Terms

## Haskell

- $\backslash x \rightarrow x + 1$
- $(\backslash x \rightarrow x + 1) 2 \quad (\rightsquigarrow 3)$
- **if True then 1 else 0**  $(\rightsquigarrow 1)$
- **(,)**  $2\ 4 \quad (\rightsquigarrow (2, 4))$
- **fst**  $(2, 4) \quad (\rightsquigarrow 2)$

 $\lambda$ -Calculus

- $\lambda x. \text{add } x\ 1$
- $(\lambda x. \text{add } x\ 1)\ 2$
- **ite True 1 0**
- **Pair 2 4**
- **fst**  $(\text{Pair } 2\ 4)$

## Remark

- ‘0’, ‘1’, ‘2’, ‘4’, ‘add’, ‘fst’, ‘ite’, ‘Pair’, and ‘True’ abbreviate more complex  $\lambda$ -terms
- supposed to “encode” behavior of **0**, **1**, **2**, **4**, **(+)**, **fst**, **if ... then ... else**, **(,)**, and **True**

## Computation

- manipulate terms to “compute” some “result”
- what are the rules?
- single rule suffices

## Example – Function Application as Usual

- given  $f(x) = x + 5$
- compute  $f(3)$  by substituting 3 for  $x$  in  $x + 5$ , written  $(x + 5)[x := 3]$

Single-Step  $\beta$ -Reduction (semi-formal definition)

- intuition: apply “function” to “argument”
- in  $\lambda$ -calculus both “functions” and “arguments” are  $\lambda$ -terms
- the  $\beta$ -rule

$$(\lambda x. s) t \rightarrow_{\beta} s[x := t]$$

substitution replacing occurrences of  $x$  by  $t$

- in words: *when applying function  $(\lambda x. s)$  to input  $t$ , replace every occurrence of  $x$  in body of function (that is,  $s$ ) by  $t$*

$\beta$

$$(\lambda x. s) t \rightarrow_{\beta} s[x := t]$$

### Examples

$$\underline{(\lambda x. x) (\lambda x. x)} \rightarrow_{\beta} x[x := \lambda x. x] = \lambda x. x$$

$$\underline{(\lambda x y. y) (\lambda x. x)} \rightarrow_{\beta} (\lambda y. y)[x := \lambda x. x] = \lambda y. y$$

$$\underline{(\lambda x y z. x z (y z)) (\lambda x. x)} \rightarrow_{\beta} \lambda y z. \underline{(\lambda x. x) z (y z)} \rightarrow_{\beta} \lambda y z. z (y z)$$

$$\underline{(\lambda x. x x) (\lambda x. x x)} \rightarrow_{\beta} \underline{(\lambda x. x x) (\lambda x. x x)} \rightarrow_{\beta} \dots$$

$\lambda x. x$       no  $\beta$ -step possible

### Renaming Variables with Permutations

- **permutation** is bijection between variables (can be represented by finite composition of swappings  $(x_n \rightleftharpoons y_n) \circ \dots \circ (x_1 \rightleftharpoons y_1)$ )
- **apply permutation**  $\pi$  to term  $t$ , written  $\pi \cdot t$

$$\pi \cdot x = \pi(x)$$

$$\pi \cdot (t u) = (\pi \cdot t) (\pi \cdot u)$$

$$\pi \cdot (\lambda x. t) = \lambda \pi(x). (\pi \cdot t)$$

### Example

- $(x \rightleftharpoons y) \cdot (\lambda x y. x (\lambda z x. z x)) = \lambda y x. y (\lambda z y. z y)$
- $((x \rightleftharpoons y) \circ (x \rightleftharpoons z)) \cdot (z y) = y x$  (order is important)

### Binders are Everywhere

- logic: for all  $\forall x. \phi(x)$ , exists  $\exists x. \psi(x)$ , ...
- calculus: derivative  $\frac{d}{dx} f$ , integral  $\int f dx$ , ...
- combinatorics: sum  $\sum_{i=1}^n g(i)$ , product  $\prod_{j=1}^n h(j)$ , ...

### Problem – Variable Capture

- consider  $\lambda x y. x$
- intended behavior: *take 2 arguments, ignore second, return first*
- evaluate  $(\lambda x y. x) y z$
- we want result  $y$ , but get  $(\lambda x y. x) y z \rightarrow_{\beta} (\lambda y. y) z \rightarrow_{\beta} z$
- clearly not intended (problem was that **free**  $y$  was **bound/captured** when substituting for  $x$ )

### Solution

- do not allow arbitrary substitution (**freshness constraints**)
- work modulo **consistent renaming** of bound variables ( **$\alpha$ -equivalence**)

### Free Variables and Freshness

- set of **free variables** of a term

$$\mathcal{F}(x) = \{x\}$$

$$\mathcal{F}(t u) = \mathcal{F}(t) \cup \mathcal{F}(u)$$

$$\mathcal{F}(\lambda x. t) = \mathcal{F}(t) \setminus \{x\}$$

- **freshness constraint**  $X \# o$  – set of variables  $X$  is **fresh for** given syntactic object  $o$  if  $X$  does not contain any free variables of  $o$
- for **single variable** write  $x \# o$  instead of  $\{x\} \# o$

### Examples

| term $t$           | $\mathcal{F}(t)$ | $x \# t$ | $\{y, z\} \# t$ |
|--------------------|------------------|----------|-----------------|
| $\lambda x. x$     | $\emptyset$      | ✓        | ✓               |
| $x y$              | $\{x, y\}$       | ✗        | ✗               |
| $(\lambda x. x) x$ | $\{x\}$          | ✗        | ✓               |
| $\lambda x. x y z$ | $\{y, z\}$       | ✓        | ✗               |

## $\alpha$ -Equivalence

- $t$  is  **$\alpha$ -equivalent** to  $u$ , written  $t \equiv_\alpha u$ , if  $u$  can be obtained by consistently renaming **bound** variables
- $\equiv_\alpha$  is equivalence relation (reflexive, symmetric, transitive)
- **from now on  $\alpha$ -equivalent terms are considered equal**
- that is, we work “modulo  $\alpha$ ”

## Examples

- $\lambda xy. x \equiv_\alpha \lambda yx. y$  since  $\lambda xy. x \equiv_\alpha (x \rightleftharpoons z) \cdot (\lambda xy. x) = \lambda zy. z \equiv_\alpha (y \rightleftharpoons x) \cdot (\lambda zy. z) = \lambda zx. z \equiv_\alpha (z \rightleftharpoons y) \cdot (\lambda zx. z) = \lambda yx. y$
- $\alpha$ -equivalence class of  $\lambda x. y x$  (remember: all elements considered the same term)

$$\{\lambda x. y x, \lambda z. y z, \lambda a. y a, \lambda x'. y x', \lambda b_1. y b_1, \dots\}$$

## Substitutions

- substitution (for terms) is function from variables to terms
- for  $\beta$  we only need substitutions replacing a single variable
- hence, we can always write  $[x := t]$  for the substitution replacing  $x$  by  $t$  and leaving all other variables unchanged

## Example

- consider  $\sigma = [x := \lambda x. x]$
- then  $\sigma(x) = \lambda x. x$  and
- $\sigma(y) = y$  for all  $y \neq x$

## Applying Substitutions to Terms

- applying substitution  $\sigma = [x := s]$  to term  $t$  is denoted by  $t\sigma$
- and defined by

$$\begin{array}{ll} x\sigma = s & \\ y\sigma = y & \text{if } x \neq y \\ (t u)\sigma = (t\sigma) (u\sigma) & \\ (\lambda y. t)\sigma = \lambda y. (t\sigma) & \text{if } y \# (x, s) \end{array}$$

- that is, only substitute for free variables
- **note:** freshness constraint can always be satisfied by renaming

## Examples

- $\sigma = [x := \lambda x. x]$
- $x\sigma = \lambda x. x$
- $y\sigma = y$
- $(\lambda x. x)\sigma = (\lambda y. y)\sigma = \lambda y. (y\sigma) = \lambda y. y$

## Single-Step $\beta$ -Reduction (formal definition)

- inductive definition of one step  $\beta$ -reduction

$$\begin{array}{ll} \frac{}{(\lambda x. t) u \rightarrow_\beta t[x := u]} \text{ (root)} & \frac{s \rightarrow_\beta t}{s u \rightarrow_\beta t u} \text{ (app-l)} \\ \frac{s \rightarrow_\beta t}{\lambda x. s \rightarrow_\beta \lambda x. t} \text{ (abs)} & \frac{s \rightarrow_\beta t}{u s \rightarrow_\beta u t} \text{ (app-r)} \end{array}$$

- intuition: if  $s$  has subterm of form  $(\lambda x. t) u$ , then replacing it by  $t[x := u]$  is a  $\beta$ -step
- we call  $(\lambda x. t) u$  a **redex** (short for **reducible expression**), and
- $t[x := u]$  its **contractum**
- $s \rightarrow_\beta^* t$  (with  $\rightarrow_\beta^*$  reflexive, transitive closure of  $\rightarrow_\beta$ ) denotes existence of sequence  $s = t_1 \rightarrow_\beta t_2 \rightarrow_\beta \dots \rightarrow_\beta t_n = t$  with  $n \geq 0$  (say “ $s$  ( $\beta$ -)reduces to  $t$ ”)
- existence of nonempty sequence (that is,  $n > 0$ ) denoted by  $s \rightarrow_\beta^+ t$

### Exercise

- consider  $\Omega = (\lambda x. x x) (\lambda x. x x)$ ,
- $K = \lambda xy. x$ ,
- $\lambda xy. y$ , and
- $I = \lambda x. x$
- reduce the following  $\lambda$ -terms

$$K \Omega$$

$$(\lambda xy. y) \Omega$$

$$I \Omega$$

### What are the Results of Computations?

- only have  $\lambda$ -terms
- thus, have to express “functions” and “results” as  $\lambda$ -terms
- as long as  $\beta$ -steps are applicable, terms are not “stable”
- one possibility: consider terms for which no  $\beta$ -step is applicable (so called “normal forms”; abbreviation NF) **result**

### Examples

- $\lambda x. x$  is in NF
- $(\lambda x. x) y$  is not in NF, since  $(\lambda x. x) y \rightarrow_{\beta} y$  (with NF  $y$ )

### Encoding – General Idea

- goal: imitate behavior
- motto: if it behaves like  $A$  (pair, list, ...) we call it  $A$

### Booleans and Conditionals

| Haskell  | $\lambda$ -Calculus                                      |
|--|--|
| • <b>True</b>  | • $\text{True} = \lambda xy. x$ “ignore second argument” |
| • <b>False</b>   | • $\text{False} = \lambda xy. y$ “ignore first argument” |
| • <b>if <math>b</math> then <math>t</math> else <math>e</math></b> | • $\text{ite} = \lambda xyz. x y z$                      |

### Examples

$$\text{ite True } x y \rightarrow_{\beta}^{+} \text{True } x y \rightarrow_{\beta}^{+} x$$

$$\text{ite False } x y \rightarrow_{\beta}^{+} \text{False } x y \rightarrow_{\beta}^{+} y$$

### Natural Numbers – Church Numerals

- define  $n$ -fold function application

$$s^0 t = t$$

$$s^{n+1} t = s (s^n t)$$

function that applies first argument  $n$ -times to second argument

- number  $n$  is represented by term  $\lambda fx. f^n x$

### Haskell vs. $\lambda$ -Calculus

| Haskell      | $\lambda$ -Calculus                        |
|--------------|--|
| • <b>0</b>   | • $0 = \lambda fx. x$                      |
| • <b>1</b>   | • $1 = \lambda fx. f x$                    |
| • <b>n</b>   | • $n = \lambda fx. f^n x$                  |
| • <b>(+)</b> | • $\text{add} = \lambda mnfx. m f (n f x)$ |
| • <b>(*)</b> | • $\text{mul} = \lambda mnfx. m (n f)$     |
| • <b>(^)</b> | • $\text{exp} = \lambda mn. n m$           |

## Pairs

|  |   |
|--|---|
| <p>Haskell</p> <ul style="list-style-type: none"> <li>• (,)</li> <li>• <code>fst</code></li> <li>• <code>snd</code></li> </ul> | <p><math>\lambda</math>-Calculus</p> <ul style="list-style-type: none"> <li>• <math>\text{Pair} = \lambda xyf. f\ x\ y</math></li> <li>• <math>\text{fst} = \lambda p. p\ \text{True}</math></li> <li>• <math>\text{snd} = \lambda p. p\ \text{False}</math></li> </ul> |
|--|---|

## Lists

|   |   |
|---|---|
| <p>Haskell</p> <ul style="list-style-type: none"> <li>• ( : )</li> <li>• <code>head</code></li> <li>• <code>tail</code></li> <li>• []</li> <li>• <code>null</code></li> </ul> | <p><math>\lambda</math>-Calculus</p> <ul style="list-style-type: none"> <li>• <math>\text{Cons} = \lambda xy. \text{Pair False (Pair } x\ y)</math></li> <li>• <math>\text{head} = \lambda z. \text{fst (snd } z)</math></li> <li>• <math>\text{tail} = \lambda z. \text{snd (snd } z)</math></li> <li>• <math>\text{Nil} = \lambda x. x</math></li> <li>• <math>\text{null} = \text{fst}</math></li> </ul> |
|---|---|

## A Fixed Point Combinator

- **note:** **combinator** is  $\lambda$ -term without free variables
- Alan Turing discovered combinator  $Y$  (sometimes  $\Theta$ ), satisfying

$$Y\ t \rightarrow_{\beta}^* t\ (Y\ t) \quad \text{for every term } t$$

$$Y\ t \rightarrow_{\beta} (\lambda f. f\ ((\lambda x f. f\ (x\ x\ f))\ (\lambda x f. f\ (x\ x\ f))\ f))\ t$$

$$\rightarrow_{\beta} t\ ((\lambda x f. f\ (x\ x\ f))\ (\lambda x f. f\ (x\ x\ f))\ t) = t\ (Y\ t)$$

- **fixed point property**
- definition of  $Y$  is (somewhat complicated)

$$Y = (\lambda x f. f\ (x\ x\ f))\ (\lambda x f. f\ (x\ x\ f))$$

## Example – Length

- recall that  $\text{length} = Y\ g$  with  $g = \lambda fx. \text{ite (null } x) 0\ (\text{add } 1\ (f\ (\text{tail } x)))$
- by fixed point property we obtain  $\text{length} \rightarrow_{\beta}^* g\ \text{length}$ , taking care of replacing additional parameter  $f$  in  $g$  by definition of  $\text{length}$

## Recursion

- Haskell function  $\text{length } x = \text{if null } x\ \text{then } 0\ \text{else } 1 + \text{length (tail } x)$
- in  $\lambda$ -calculus  $\text{length} \stackrel{?}{=} \lambda x. \text{ite (null } x) 0\ (\text{add } 1\ (\text{length (tail } x)))$

- **problem:**  $\text{length}$  is not allowed to occur on right-hand side
- try to cope by adding additional argument

$$\text{length}' = \lambda fx. \text{ite (null } x) 0\ (\text{add } 1\ (f\ (\text{tail } x)))$$

- idea: at some point  $f$  should be replaced by  $\text{length}$  again
- partial solution:

$$\text{length} = Y\ \text{length}'$$

- missing: find appropriate  $Y$

## Exercise Preparation – Instantiating Type Classes

general schema for turning type  $T$  into instance of type class  $C$

```
instance C T where
  -- implementations of class functions
```

## Example – Equality for User-Defined Type

- consider type `data YNM = Yes | No | Maybe`
- instance declaration

```
instance Eq YNM where
  Yes == Yes = True
  No == No = True
  Maybe == Maybe = True
  _ == _ = False
```

## Homework (for November 30th)

1. Read the [lecture notes on the lambda calculus](#) until Section 5.
2. Encode the Haskell function `boolToInt :: Bool -> Int`

```
boolToInt True  = 1
boolToInt False = 0
```

as  $\lambda$ -term and step-wise compute the NF of the  $\lambda$ -term encoding `boolToInt True`.
3. Give a combinator  $E$  (“eraser”) that satisfies  $E t \rightarrow_{\beta}^* E$  for arbitrary  $\lambda$ -terms  $t$ .  
**Hint:** Fix  $E = A A$  and try to come up with an appropriate  $A$ .
4. Given the following Haskell type for  $\lambda$ -terms

```
type Id = String
data Term = Var Id | App Term Term | Abs Id Term
```

implement a function `redexes :: Term -> [Term]` that computes the list of all redexes occurring in a term.

## Homework (for November 30th, continued)

5. Write a `Show` class instance for the `Term` type of Exercise 4 that produces pretty output conforming to the conventions of slide 6.  
**Example:**

```
show (App (App (Var "x") (Abs "y" (Var "y")))) (Var "z"))
= "x (\y. y) z"
```

**Hint:** Consult the official [documentation](#) of `Show`, `showsPrec`, `showString`, and `showParen`.
6. Implement a function

```
subst :: String -> Term -> Term -> Term
```

such that `subst x s t` computes `t[x := s]` as given on slide 15. (Bonus: satisfy arising freshness constraints on the fly.)