

Functional Programming

Christian Sternagel Harald Zankl Evgeny Zuenko

Department of Computer Science
University of Innsbruck

WS 2017/2018

Lecture 5



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, implementing a type checker, 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 λ -Calculus
- Encoding Data Types

Origin

- search for general framework in which every algorithm can be defined
- “universal language” (concerning computation)
- in 1936, Alonzo Church introduced the λ -Calculus in his paper *An Unsolvable Problem of Elementary Number Theory*, *AJM* **58**(2), pages 345–363, [doi:10.2307/2371045](https://doi.org/10.2307/2371045)
- in 1937, Alan Turing introduced Turing Machines in his paper *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)
- later it was shown that both models of computation are equivalent
- that is, Turing-complete is the same as definable in the λ -Calculus
- λ -Calculus is underlying much of functional programming

Syntax – λ -Terms

- grammar

$t ::= x$	variable/atom
$(t t)$	application
$(\lambda x. t)$	(lambda) abstraction

- set of lambda terms is denoted Λ

Examples

$$\begin{aligned}
 & (\lambda x. y) \\
 & (\lambda x. (\lambda y. x)) \\
 & (\lambda x. (\lambda y. (\lambda z. ((x z) (y z))))) \\
 & (\lambda x. ((\lambda y. (\lambda z. (z y))) x))
 \end{aligned}$$

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 $(\lambda x. x) z$
- applications associate to the left
- nested abstractions associate to the right and variables are combined

Examples (using Conventions)

$$\lambda x. y$$

$$\lambda xy. x$$

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

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

Note

- nested lambdas are “functions with multiple arguments”
- e.g., $\lambda xyz. t$ is a function taking 3 arguments

λ -Terms and Haskell

Haskell

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

λ -Calculus

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

Remark

- '0', '1', '2', '4', 'add', 'fst', 'if', 'Pair', and 'True' are abbreviations for more complex λ -terms
- supposed to "encode" the behavior of $0, 1, 2, 4, (+), \dots$

Computation

- manipulate terms to “compute” some “result”
- what are the rules?
- it turns out that a single rule is enough

The β -Rule (informal definition)

- intuition: apply “function” to “argument”
- in λ -Calculus both “functions” and “arguments” are λ -terms
- the β -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*

The β -Rule

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

Examples

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

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

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

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

$\lambda x. x$ **no β -step possible**

Problem – Variable Capture

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

Solution

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

Renaming Variables with Permutations

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

$$\pi \bullet x = \pi(x)$$

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

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

Example

- $(x \rightleftharpoons y) \bullet (\lambda x y. x (\lambda z x. z x)) = \lambda y x. y (\lambda z y. z y)$
- $(x \rightleftharpoons z, y \rightleftharpoons x, z \rightleftharpoons y) \bullet (z y) = y x$

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 , e.g., if o is term then $X \# o$ iff $X \cap \mathcal{F}(o) = \emptyset$
- 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	✓	✓
$x y$	$\{x, y\}$	✗	✗
$(\lambda x. x) x$	$\{x\}$	✗	✓
$\lambda x. x y z$	$\{y, z\}$	✓	✗

α -Equivalence

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

Examples

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

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

Substitutions

- substitution (for terms) is function from variables to terms
- 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

$$x\sigma = s$$

$$y\sigma = y \quad \text{if } x \neq y$$

$$(t u)\sigma = (t\sigma) (u\sigma)$$

$$(\lambda y. t)\sigma = \lambda y. (t\sigma) \quad \text{if } y \# (x, s)$$

- that is, bound variables are **not** substituted
- **note:** freshness constraint can always be satisfied by renaming

Examples

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

The β -Rule (formal definition)

- inductive definition of one step β -reduction

$$\frac{}{(\lambda x. t) u \rightarrow_{\beta} t[x := u]} \text{ (root)} \qquad \frac{s \rightarrow_{\beta} t}{s u \rightarrow_{\beta} t u} \text{ (app-l)}$$

$$\frac{s \rightarrow_{\beta} t}{\lambda x. s \rightarrow_{\beta} \lambda x. t} \text{ (abs)} \qquad \frac{s \rightarrow_{\beta} t}{u s \rightarrow_{\beta} u t} \text{ (app-r)}$$

- in words: if s has subterm of form $(\lambda x. t) u$, then replacing it by $t[x := u]$ is a β -step
- we call $(\lambda x. t) u$ a **redex** (short for *reducible expression*), and
- $t[x := u]$ its **contractum**
- $s \rightarrow_{\beta}^* t$ denotes a sequence $s = t_1 \rightarrow_{\beta} t_2 \rightarrow_{\beta} \cdots \rightarrow_{\beta} t_n = t$ with $n \geq 0$ (s (β -)reduces to t)
- a nonempty sequence (that is, $n > 0$) is denoted by $s \rightarrow_{\beta}^+ t$

Exercise

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

$$K \Omega$$
$$K_* \Omega$$
$$I \Omega$$

What are the Results of Computations?

- we do only have λ -terms
- thus, we have to express “functions” and “results” as λ -terms
- as long as β -steps are applicable, terms are not “stable”
- one possibility: consider terms for which no β -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)

Booleans and Conditionals

Haskell

- `True`
- `False`
- `if b then t else e`

λ -Calculus

- `True` = $\lambda xy. x$ “ignore second argument”
- `False` = $\lambda xy. y$ “ignore first argument”
- `if` = $\lambda xyz. x y z$

Examples

$$\begin{array}{ll} \text{if True } x y & \rightarrow_{\beta}^{+} \text{ True } x y \rightarrow_{\beta}^{+} x \\ \text{if False } x y & \rightarrow_{\beta}^{+} \text{ False } x y \rightarrow_{\beta}^{+} y \end{array}$$

Natural Numbers – Church Numerals

- define n -fold application of “function”

$$s^0 t = t$$

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

function that applies first argument n -times to second argument

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

Haskell vs. λ -Calculus

Haskell

- 0
- 1
- n
- (+)
- (*)
- (\wedge)

λ -Calculus

- $0 = \lambda f x. x$
- $1 = \lambda f x. f x$
- $n = \lambda f x. f^n x$
- $\text{add} = \lambda m n f x. m f (n f x)$
- $\text{mul} = \lambda m n f. m (n f)$
- $\text{exp} = \lambda m n. n m$

Pairs

Haskell

- `(,)`
- `fst`
- `snd`

λ -Calculus

- $\text{Pair} = \lambda xyf. f\ x\ y$
- $\text{fst} = \lambda p. p\ \text{True}$
- $\text{snd} = \lambda p. p\ \text{False}$

Lists

Haskell

- `(:)`
- `head`
- `tail`
- `[]`
- `null`

λ -Calculus

- $\text{Cons} = \lambda xy. \text{Pair}\ \text{False}\ (\text{Pair}\ x\ y)$
- $\text{head} = \lambda z. \text{fst}\ (\text{snd}\ z)$
- $\text{tail} = \lambda z. \text{snd}\ (\text{snd}\ z)$
- $\text{Nil} = \lambda x. x$
- $\text{null} = \text{fst}$

Recursion

- Haskell function

```
length x = if null x then 0
          else 1 + length (tail x)
```

- in λ -Calculus

$$\text{length} \stackrel{?}{=} \lambda x. \text{if } (\text{null } x) \text{ } 0 \text{ (add 1 (length (tail } x))\text{))}$$

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

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

- idea: at some point f should be replaced by length again
- partial solution:

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

- missing: find appropriate Y

The Y-Combinator

- **note:** **combinator** is λ -term without free variables
- Haskell Curry found combinator Y , satisfying

$$Y t \equiv_{\beta} t (Y t) \quad \text{for every term } t$$

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

$\beta \leftarrow$

arbitrarily many β -steps
in arbitrary direction

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

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

Example – Length

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

Exercises (for November 17th)

1. Read the [lecture notes on the lambda calculus](#) until Section 5.
2. Use the conventions to simplify $\lambda x. (\lambda y. (\lambda z. ((z (x y)) x)))$ and drop the conventions in the term $\lambda a b c d. a b c d$.
3. Consider $F = \lambda f x y. f y x$. What does F do? What does $F (\lambda x y. x)$ do? Reduce $F (\lambda x y. x)$ to NF.
4. Using `\` and `union` from `Data.List` as well as the types

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

implement functions `vars` and `fvs` (both of type `Term -> [Id]`) computing the set (in the form of a list without duplicates) of all variables and free variables occurring in a term, respectively.
5. Implement a function `freshName` that, given a variable `x` and a list of variables `ys`, computes a pair `(x', f)` where `x'` is fresh for `ys` and `f :: Term -> Term` replaces all occurrences of `x` in a term by `x'`.
6. With the help of `vars`, `fvs`, and `freshName`, implement a function `subst :: String -> Term -> Term -> Term`, where `subst x s t` computes $t[x := s]$, satisfying the freshness constraint.