universität innsbruck

WS 2018/2019

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 λ-Calculus

- Encoding Data Types

Origin

- search for general framework in which every algorithm can be defined
- "universal language" (for computation)
- 1936 – Alonzo Church introduces λ-calculus in *An Unsolvable Problem of Elementary Number Theory*, AJM **58**(2), pages 345–363, doi: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
- and shows both models of computation equivalent
- that is, Turing-complete is same as definable in λ-calculus
- λ-calculus closer to functional programming

Syntax – λ-Terms

- grammar

$$\begin{array}{rcll} t & ::= & x & \text{variable/atom} \\ & | & (t\ t) & \text{application} \\ & | & (\lambda x.\, t) & \text{(lambda) abstraction} \end{array}$$

- 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)

$$\lambda x.\, y$$

$$\lambda xy.\, x$$

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

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

Note

- think of nested lambdas as "functions with multiple arguments"
- e.g., $\lambda xyz.\, t$ is function taking 3 arguments

Haskell vs. λ-Terms

Haskell		λ-Calculus
`\x -> x + 1`		$\lambda x.\,\text{add}\ x\ 1$
`(\x -> x + 1) 2`	$(\rightsquigarrow 3)$	$(\lambda x.\,\text{add}\ x\ 1)\ 2$
`if True then 1 else 0`	$(\rightsquigarrow 1)$	$\text{ite}\ \text{True}\ 1\ 0$
`(,) 2 4`	$(\rightsquigarrow$ `(2, 4)`$)$	$\text{Pair}\ 2\ 4$
`fst (2, 4)`	$(\rightsquigarrow 2)$	$\text{fst}\ (\text{Pair}\ 2\ 4)$

Remark

- '0', '1', '2', '4', 'add', 'fst', 'ite', 'Pair', and 'True' abbreviate more complex λ-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 β-Reduction (semi-formal definition)

- intuition: apply "function" to "argument"
- in λ-calculus both "functions" and "arguments" are λ-terms
- the β-rule

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

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*

β

$$(\lambda x.\, s)\ t \quad \to_\beta \quad s[x := t]$$

Examples

$$\begin{array}{rcll}
\underline{(\lambda x.\, x)\ (\lambda x.\, x)} & \to_\beta & x[x := \lambda x.\, x] & = \quad \lambda x.\, x \\
\underline{(\lambda xy.\, y)\ (\lambda x.\, x)} & \to_\beta & (\lambda y.\, y)[x := \lambda x.\, x] & = \quad \lambda y.\, y \\
\underline{(\lambda xyz.\, x\ z\ (y\ z))\ (\lambda x.\, x)} & \to_\beta & \lambda yz.\, \underline{(\lambda x.\, x)\ z}\ (y\ z) & \to_\beta \quad \lambda yz.\, z\ (y\ z) \\
\underline{(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)} & \to_\beta & \underline{(\lambda x.\, x\ x)\ (\lambda x.\, x\ x)} & \to_\beta \quad \cdots \\
\lambda x.\, x & & \text{no } \beta\text{-step possible} &
\end{array}$$

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 $\lambda xy.\, x$
- intended behavior: *take 2 arguments, ignore second, return first*
- evaluate $(\lambda xy.\, x)\ y\ z$
- we want result y, but get $\underline{(\lambda xy.\, x)\ y}\ z \to_\beta \underline{(\lambda y.\, y)\ z} \to_\beta z$
- 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 (α-equivalence)

Renaming Variables with Permutations

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

$$\begin{aligned} \pi \cdot x &= \pi(x) \\ \pi \cdot (t\ u) &= (\pi \cdot t)\ (\pi \cdot u) \\ \pi \cdot (\lambda x.\, t) &= \lambda \pi(x).\, (\pi \cdot t) \end{aligned}$$

Example

- $(x \rightleftharpoons y) \cdot (\lambda xy.\, x\ (\lambda zx.\, z\ x)) = \lambda yx.\, y\ (\lambda zy.\, z\ y)$
- $((x \rightleftharpoons y) \circ (x \rightleftharpoons z)) \cdot (z\ y) = y\ x$ (order is important)

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 $X \sharp o$ – 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 $x \sharp o$ instead of $\{x\} \sharp o$

Examples

term t	$\mathcal{F}(t)$	$x \sharp t$	$\{y, z\} \sharp t$
$\lambda x.\, x$	$\varnothing$	✔	✔
$x\ y$	$\{x, y\}$	✘	✘
$(\lambda x.\, x)\ x$	$\{x\}$	✘	✔
$\lambda x.\, x\ y\ z$	$\{y, z\}$	✔	✘

α-Equivalence

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

Examples

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

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

Substitutions

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

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

- 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 β-Reduction (formal definition)

- inductive definition of one step β-reduction

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

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

- intuition: *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
- $s \to_\beta^* t$ (with $\to_\beta^*$ reflexive, transitive closure of $\to_\beta$) denotes existence of sequence $s = t_1 \to_\beta t_2 \to_\beta \cdots \to_\beta t_n = t$ with $n \geq 0$
  (say "s (β-)reduces to t")
- existence of nonempty sequence (that is, $n > 0$) denoted by $s \to_\beta^+ t$

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 λ-terms

$$K\ \Omega$$
$$(\lambda xy.\, y)\ \Omega$$
$$I\ \Omega$$

What are the Results of Computations?

- only have λ-terms
- thus, 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 $(\lambda x.\, x)\ y \to_\beta y$ (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

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

λ-Calculus

- True $= \lambda xy.\, x$ "ignore second argument"
- False $= \lambda xy.\, y$ "ignore first argument"
- ite $= \lambda xyz.\, x\ y\ z$

Examples

$$\begin{array}{llll} \text{ite True } x\ y & \to_\beta^+ & \text{True } x\ y & \to_\beta^+ \quad x \\ \text{ite False } x\ y & \to_\beta^+ & \text{False } x\ y & \to_\beta^+ \quad y \end{array}$$

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 $\lambda f x.\ f^n\ x$

Haskell vs. λ-Calculus

Haskell	λ-Calculus
`0`	$0 = \lambda f x.\ x$
`1`	$1 = \lambda f x.\ f\ x$
`n`	$n = \lambda f x.\ f^n\ x$
`(+)`	$\text{add} = \lambda m n f x.\ m\ f\ (n\ f\ x)$
`(*)`	$\text{mul} = \lambda m n f.\ m\ (n\ f)$
`(^)`	$\text{exp} = \lambda m n.\ n\ m$

Pairs

Haskell

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

λ-Calculus

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

Lists

Haskell

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

λ-Calculus

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

Recursion

- Haskell function

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

- in λ-calculus

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

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

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

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

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

- missing: find appropriate Y

A Fixed Point Combinator

- **note:** combinator is λ-term without free variables
- Alan Turing discovered combinator Y (sometimes Θ), satisfying

$$\mathsf{Y}\ t \to_\beta^* t\ (\mathsf{Y}\ t) \qquad \text{for every term } t$$

$$\mathsf{Y}\ t \to_\beta (\lambda f.\, f\ ((\lambda x f.\, f\ (x\ x\ f))\ (\lambda x f.\, f\ (x\ x\ f))\ f))\ t$$
$$\to_\beta t\ ((\lambda x f.\, f\ (x\ x\ f))\ (\lambda x f.\, f\ (x\ x\ f))\ t) = t\ (\mathsf{Y}\ t)$$

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

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

Example – Length

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

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 λ-term and step-wise compute the NF of the λ-term encoding `boolToInt True`.
3. Give a combinator E ("eraser") that satisfies $E\,t \to_\beta^* E$ for arbitrary λ-terms t.
   Hint: Fix $E = A\,A$ and try to come up with an appropriate A.
4. Given the following Haskell type for λ-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 $\texttt{t}[\texttt{x} := \texttt{s}]$ as given on slide 15.
   (Bonus: satisfy arising freshness constraints on the fly.)