

LiU-FP2016: Lecture 2

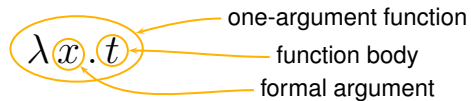
The Untyped λ -Calculus: Introduction

Henrik Nilsson
University of Nottingham, UK

LIU-FP2016: Lecture 2 – p.1/15

Key Idea

λ -abstraction (or anonymous function):



Multiple arguments handled by “returning” lambda abstractions that then are applied to further arguments: **Currying**.

LIU-FP2016: Lecture 2 – p.4/15

Exercise

In the following:

- Which variables are free and which are bound?
- Which terms are open and which are closed?

- (a) x (d) $\lambda x.\lambda y.x y$
 (b) $\lambda x.x$ (e) $(\lambda x.x) x$
 (c) $\lambda x.y$ (f) $\lambda x.\lambda y.(\lambda x.x y) (\lambda z.x y)$

LIU-FP2016: Lecture 2 – p.7/15

The λ -Calculus: What is it? (1)

- Pure notion of effective computation procedure: **all** computation reduced to function definition and application.
- Invented in the 1920s by Alonzo Church.
- Cf. other formalisations of the notion of effective computation; e.g., the Turing machine.
- The λ -calculus and Turing Machines are equivalent in that they capture the exact same notion of what “computation” means.

LIU-FP2016: Lecture 2 – p.2/15

Syntax

$t \rightarrow$		terms:
	x	variable
	$\lambda x.t$	abstraction
	$t t$	application

Note:

- x is the syntactic category of variables. We will use actual names like x, y, z, u, v, w, \dots
- λ -abstractions often named for convenience. E.g. $I \equiv \lambda x.x$. **Just an abbreviation!**
So e.g. $F \equiv \lambda x.(\dots F \dots)$ **not** valid def. Why?

LIU-FP2016: Lecture 2 – p.5/15

Operational Semantics (1)

Sole means of computation: **β -reduction** or **function application**:

$$(\lambda x.t_1) t_2 \rightarrow_{\beta} [x \mapsto t_2]t_1$$

where

$$[x \mapsto t_2]t_1$$

means “term t_1 with all **free** occurrences of x (with respect to t_1) replaced by t_2 .”

Subtle problems concerning **name clashes** will be considered later.

LIU-FP2016: Lecture 2 – p.8/15

The λ -Calculus: What is it? (2)

- The Church-Turing Hypothesis: The λ -calculus, Turing Machines, etc. coincides with our intuitive understanding of what “computation” means.
- The λ -calculus is important because it is at once:
 - very simple, yet in essence a practically useful programming language
 - mathematically precise, allowing for formal reasoning.

LIU-FP2016: Lecture 2 – p.9/15

Scope

- An **occurrence** of x is **bound** if it occurs in the body t of a λ -abstraction $\lambda x.t$.
- A non-bound occurrence is **free**.
- A λ -term with **no free** variables is called **closed**. Otherwise **open**.
- A closed λ -term is called a **combinator**.

LIU-FP2016: Lecture 2 – p.11/15

Operational Semantics (2)

A term that can be β -reduced is called a **(β -)redex**.

Exercise: Underline the redexes in

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

LIU-FP2016: Lecture 2 – p.11/15

Programming In the λ -Calculus

How can such a simple language express arbitrary computations?

Nothing that looks like arithmetic, or conditionals, and seems not even recursion allowed?

To make it plausible that the λ -calculus indeed is a general notion of computation, we will see how to express:

- Booleans
- Arithmetic
- Recursion

LIJ-PP2016: Lecture 2 – p.10/15

Church Numerals (1)

Idea: The natural number n is represented by a function that applies its first argument n times to its second argument.

$$\begin{aligned}C_0 &\equiv \lambda s.\lambda z.z \\C_1 &\equiv \lambda s.\lambda z.s\ z \\C_2 &\equiv \lambda s.\lambda z.s\ (s\ z) \\C_3 &\equiv \lambda s.\lambda z.s\ (s\ (s\ z))\end{aligned}$$

Etc.

LIJ-PP2016: Lecture 2 – p.13/15

Church Booleans

True, false, and conditional:

$$\begin{aligned}T &\equiv \lambda t.\lambda f.t \\F &\equiv \lambda t.\lambda f.f \\IF &\equiv \lambda l.\lambda m.\lambda n.l\ m\ n\end{aligned}$$

Exercise: Evaluate $IF\ T\ v\ w$

Logical connectives:

$$\begin{aligned}AND &\equiv \lambda b.\lambda c.b\ c\ F \\OR &\equiv \lambda b.\lambda c.b\ T\ c \\NOT &\equiv \lambda b.b\ F\ T\end{aligned}$$

LIJ-PP2016: Lecture 2 – p.11/15

Church Numerals (2)

Operations:

$$\begin{aligned}SUCC &\equiv \lambda n.\lambda s.\lambda z.s\ (n\ s\ z) \\PLUS &\equiv \lambda m.\lambda n.\lambda s.\lambda z.m\ s\ (n\ s\ z) \\TIMES &\equiv \lambda m.\lambda n.\lambda s.m\ (n\ s) \\POWER &\equiv \lambda m.\lambda n.m\ n \\ISZERO &\equiv \lambda m.m\ (\lambda x.F)\ T\end{aligned}$$

LIJ-PP2016: Lecture 2 – p.14/15

Pairs

If we can represent pairs, then we can represent any kind of compound data:

$$\begin{aligned}PAIR &\equiv \lambda f.\lambda s.\lambda b.b\ f\ s \\FST &\equiv \lambda p.p\ T \\SND &\equiv \lambda p.p\ F\end{aligned}$$

LIJ-PP2016: Lecture 2 – p.12/15

Church Numerals (3)

Subtraction is more intricate. Let us consider the predecessor function:

$$\begin{aligned}ZZ &\equiv PAIR\ C_0\ C_0 \\SS &\equiv \lambda p.PAIR\ (SND\ p)\ (SUCC\ (SND\ p)) \\PRED &\equiv \lambda m.FST\ (m\ SS\ ZZ)\end{aligned}$$

Idea: SS maps (m, n) to $(n, n + 1)$. Iterating SS n times on $(0, 0)$ means that the first component of the result is $n - 1$.

LIJ-PP2016: Lecture 2 – p.15/15