

# Lecture 2: Formal Definition of $\lambda$ -Calculus

Ankush Das

January 26, 2024

## 1 Introduction

In the last lecture, we saw a toy language: LL1 and studied its syntax and semantics. In this lecture, we will return to  $\lambda$ -calculus and study its syntax and semantics.

## 2 The $\lambda$ -Calculus

### 2.1 Syntax

Since the essence of  $\lambda$ -calculus is functions, the syntax of  $\lambda$ -calculus is defined only using 3 expressions:

$$\text{Expressions } e ::= \lambda x.e \mid e_1 e_2 \mid x$$

The first expression  $\lambda x.e$  defines a function with parameter  $x$  and body  $e$ . The second expression simply applies function  $e_1$  to the argument  $e_2$ . The last expression is a variable which is essential to refer to the parameter in the body of the expression.

**Some Examples** Now that we've seen the grammar, let's look at some examples of expressions in  $\lambda$ -calculus.

- $\lambda x.x$ : the simplest example is that of an identity function. The body of the expression is just  $x$  meaning that the function just returns its parameter.
- $\lambda x.\lambda y.x$ : this function takes two parameters  $x$  and  $y$  but only returns the first one (and throws away the second one). We can similarly define  $\lambda x.\lambda y.y$ . Soon, we will see how these expressions represent booleans.

### 2.2 Semantics

Now that we have seen some examples, let's try to define how these expressions can be evaluated. There are two standard ways of defining a semantics: (i) small-step semantics and (ii) big-step semantics.

**Small-Step Semantics** This defines a single step of evaluation. This is usually represented as  $e \mapsto e'$ , meaning expression  $e$  reduces to expression  $e'$  in a *single step*. Now, we define the rules for  $\lambda$ -calculus. To do that, we need to define another judgment  $e$  **value** to define that  $e$  is a value and can no longer be evaluated further. Formally, for every expression  $e$ , either  $e \mapsto e'$  for some  $e'$  or  $e$  **value**, meaning either expression  $e$  steps to another expression or is a value.

$$\frac{}{\lambda x.e \text{ value}} \lambda\text{-V} \qquad \frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \text{APP-L} \qquad \frac{e_1 \text{ value} \quad e_2 \mapsto e'_2}{e_1 e_2 \mapsto e_1 e'_2} \text{APP-R}$$

First,  $\lambda$ -expressions are values. There is no way to evaluate a function unless it has been applied to some arguments. Side note: A slogan at CMU "Functions are Values!" comes from here!! Next, for function applications, we first evaluate the left hand side (chosen arbitrarily) and then the right

hand side. The App-L rule is responsible for evaluating the lhs and once  $e_1$  becomes a value, we can evaluate the rhs using rule App-R. The most important step here comes next.

$$\frac{e' \text{ value}}{(\lambda x.e) e' \mapsto [e'/x]e} \text{ APP-S}$$

Once the argument becomes a value too, the next step is to substitute the argument  $e'$  for parameter  $x$  in the function body  $e$ . Substitution means syntactically replacing every occurrence of  $x$  with  $e'$ .

Note: A technical term for small-step is also  $\beta$ -conversion or  $\beta$ -reduction. I will explain this more a little later.

**Big-Step Semantics** In contrast to small-step semantics which only describes a single step, big-step semantics describes what an expression evaluates to, no matter how many steps it takes. This is defined using the judgment  $e \Downarrow v$ , meaning expression  $e$  evaluates to value  $v$ . So, how are the rules defined?

$$\frac{}{\lambda x.e \Downarrow \lambda x.e} \lambda\text{-V} \qquad \frac{e_1 \Downarrow \lambda x.e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v} \text{ APP}$$

$\lambda$ -expressions are values, so they just evaluate to themselves. For function applications, we first evaluate  $e_1$  to  $\lambda x.e$ , then we evaluate  $e_2$  to  $v_2$ . We then substitute  $v_2$  for  $x$  in  $e$  which is then evaluated to  $v$ . Note that this semantics rule is really a combination of the rules presented in the small-step semantics.

### 3 Type Safety

We conclude this lecture by discussing type safety theorems for  $\lambda$ -calculus. Type safety is usually proved using two theorems: *progress* and *preservation*. These theorems are at the foundation of any programming language and are generally used to determine if a programming language is well-defined and sound. Since we have not introduced types into the language, we will only look at a limited version of the progress theorem.

**Theorem 1 (Progress).** *For all expressions  $e$  in  $\lambda$ -calculus such that  $e$  closed, either  $e \mapsto e'$  for some expression  $e'$  or  $e$  value.*

To understand this theorem, we first need to define what is a closed expression. Informally, a closed expression does not have any free variables. The set of free variables of an expression is defined as:

$$\frac{FV(e) = S}{FV(\lambda x.e) = S \setminus \{x\}} \lambda\text{-FV} \qquad \frac{}{FV(x) = \{x\}} \text{VAR-FV} \qquad \frac{FV(e_1) = S_1 \quad FV(e_2) = S_2}{FV(e_1 e_2) = S_1 \cup S_2} \text{APP-FV}$$

Finally, an expression is closed if it has no free variables.

$$\frac{FV(e) = \emptyset}{e \text{ closed}} \text{CLOSED}$$

*Proof.* We only have three expressions in  $\lambda$ -calculus

$$e ::= \lambda x.e \mid e_1 e_2 \mid x$$

This proof proceeds by *structural induction on the structure of the expression*. There are three cases:

- $e = \lambda x.e$ . In this case, we simply use  $\lambda$ -V rule to show this is a value.

$$\frac{}{\lambda x.e \text{ value}} \lambda\text{-V}$$

- $e = x$ . In this case, we cannot derive  $e$  is closed. Hence, the theorem holds vacuously.

- $e = e_1 e_2$ . Now, we appeal to the inductive hypothesis. We can assume the progress theorems hold for  $e_1$  and  $e_2$ .

Now, we can subcase on the outcome of the inductive hypothesis. Assume  $e_1 \mapsto e'_1$ . In this subcase, we can apply APP-L rule

$$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \text{ APP-L}$$

Hence, the progress theorem holds for  $e$  as  $e \mapsto e'_1 e_2$ .

Assume that  $e_1$  **value**. Now, we appeal to the inductive hypothesis for  $e_2$ . Assume that  $e_2 \mapsto e'_2$ . Now, we can apply APP-R rule

$$\frac{e_1 \text{ value} \quad e_2 \mapsto e'_2}{e_1 e_2 \mapsto e_1 e'_2} \text{ APP-R}$$

Again, the progress theorem holds for  $e$  as  $e \mapsto e_1 e'_2$ .

Now, assume that  $e_1$  **value** and  $e_2$  **value**. In this case, we can apply the APP-S rule.

$$\frac{e_1 \text{ value} \quad e_1 = \lambda x.e \quad e_2 \text{ value}}{(\lambda x.e) e_2 \mapsto [e_2/x]e} \text{ APP-S}$$

Again, the progress theorem holds for  $e$  as  $e \mapsto [e_2/x]e$ .

Hence, the progress theorem holds in all possible cases. □