Lecture 1: Introduction to Programming Languages: λ-Calculus

Ankush Das

January 22, 2024

1 Introduction

Let's begin with an age-old question: what is a programming language? Wikipedia states that "A programming language is a system of notation for writing computer programs.". In this course, we will explore how a programming language is so much more than a mere tool for writing programs.

Before delving into this exploration, let's return to another age-old question: what is the essence of programming? I claim that the most basic abstraction that a programming language provides capturing its core is the notion of *function*. In today's lecture, we will study this most basic abstraction and how powerful it is. We will study functions in the context of a simple and minimal programming language: the λ-calculus. This will allow us to study this concept in its full depth.

2 The λ-Calculus

The λ-calculus is one of the most foundational languages with a rich history. One of the (many) reasons it is special is due to the Church-Turing Thesis which establishes an equivalence between the λ-calculus and Turing machines, stating that any function that can be implemented using a Turing machine can be effectively computed in the λ-calculus.

Now, let's explore the λ-calculus in more detail. As we noted before, the main abstraction this language provides is *function*. How are these functions defined? Let's see a mathematical function first.

$$f(x) = x + 20 \qquad g(y) = y \times y$$

This describes how these functions operate. In the λ-calculus, these functions are defined as follows:

$$f = \lambda x.x + 20 \qquad g = \lambda y.y \times y$$

The general λ expression is written as $\lambda x.e$ where e is the function body.

The other general expression in the language is function application, written formally as $f\,e$ which means calling the function f on the expression e. We will see more examples soon.

3 Definition of λ-Calculus

Now, we will try to answer the fundamental question we asked at the start of the lecture: what is a programming language and how do we define it? I like to think of a programming language as a mathematical object (similar to other objects like circle, triangle, polynomial, etc.) which can be defined using the following two components:

- ***Syntax***: How are programs in this language written? This is similar to defining how to construct a polynomial.
- ***Semantics***: How do programs behave? This is similar to describing how to evaluate a polynomial.

In the remaining lecture, we will study the syntax and semantics of the λ-calculus.

3.1 Syntax

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

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

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 λ-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.

3.2 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 λ-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.

$$\frac{}{\lambda x.e\ \mathsf{value}}\ \lambda\text{-V} \qquad \frac{e_1 \mapsto e_1'}{e_1\ e_2 \mapsto e_1'\ e_2}\ \textsc{App-L} \qquad \frac{e_1\ \mathsf{value} \qquad e_2 \mapsto e_2'}{e_1\ e_2 \mapsto e_1\ e_2'}\ \textsc{App-R}$$

First, λ-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.

$$\frac{e'\ \mathsf{value}}{(\lambda x.e)\ e' \mapsto [e'/x]e}\ \textsc{App-S}$$

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 occurence of x with e'.

Note: A technical term for small-step is also β-conversion or β-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?

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

λ-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.