Lecture Notes on
Overview and The Lambda Calculus

15-814: Types and Programming Languages
Jan Hoffmann and Frank Pfenning

Lecture 1
Tuesday, August 27, 2024

1 Overview

We start with a motivation of the course contents and then informally discus the (untyped) λ-Calculus.

What is a Programming Language? Before we can talk about how we study programming languages, we first should define what a programming language is. If you think about the question then you may consider a definition such as *the text files accepted by a compiler or interpreter*. However, this definition is kicking the can down the road because then you would have to define what a compiler or interpreter is. Even if we had such a definition, the resulting language definition would be somewhat unsatisfactory: Handing somebody a compiler is not a good way to describe a programming language.

A better definition would be to say that *a programming language is defined by a language standard*. This is rather unspecific but the right idea. So what is a language standard? Most commonly, it is a document written in a natural language. If you ever read a language standard (like the C11 Standard) then you know that it is often unclear and sometimes underspecified. This is not surprising since natural language is all too often imprecise. Also, does a programming language even exist if it is not implemented?

In this course, our point of view is that a programming language is a mathematical object. It is not defined by an implementation and not ambiguous like a language standard written in a language like English. Instead,

a programming language is formally defined by its static and dynamic semantics, which we usually define by rule induction. The static semantics defines the set of programs (or, more generally, the set of expressions) and the dynamic semantics defines what the result is of running (or evaluating) a program.

Why Study Programming Languages? The are many good reasons to study programming languages. We want to understand both existing and future programming languages and be able to compare them in a systematic way. If we start a new project then we want to be able to make an informed decision about the right programming language to select for the project. We might even want to design a new programming language and should then be in a position to avoid past mistakes. Finally, understanding programming languages helps to become a better programmer since you will be able to better pick the right tool (or language feature) for a given implementation task.

How to Study Programming Languages? In this course, we will not study or compare popular languages. For one thing, we don't want to reinforce or repeat common mistakes in language design. For another thing, popular languages are complex and difficult to describe. Finally, the popularity of languages changes over time and we aim to focus on knowledge that enables you to understand existing and future languages. Similarly, we will not talk about so called paradigms like *imperative* or *functional* programming. At best, such paradigms describe a programming style. You can write functional programs in an imperative language and all practical functional languages exhibit imperative programs. Therefore, using paradigms for classifying languages makes little sense. But what can we do instead?

Most practical languages have the same expressivity; we say they are *Turing complete*. This means we can implement the same set of functions.[1] So how do programming languages differ? Why are we usually not using assembly or machine languages to write programs? The reason is that a programming language provides mechanisms to systematically structure programs. The key is that these mechanisms are provided by abstractions that are enforced by the language. *The study of programming languages focuses therefore on the study of abstraction and composition*. Abstractions hide information, restrict interaction, and, in return, guarantee certain properties that we can rely on when implementing programs. Composition is about combining

[1] We will make this more precise in the next weeks.

programs without breaking abstractions. With this view, programming languages are interesting if they provide strong abstractions and are therefore restrictive.

A common abstraction found in most language is that of a function. The only way to interact with a function is to call it by providing an argument. You cannot jump into an arbitrary position in the body of the implementation of the function and only execute a part of it (like you could in an assembly language).

Type theory provides a systematic way of studying and characterizing such abstractions. Types classify different forms of data and computations. By studying types, we focus on the atomic parts of programming languages and distinguish languages based on the abstractions they provide.

2 The λ-Calculus

This course is about the principles of programming language design, many of which derive from the notion of *type*. Nevertheless, we will start by studying an exceedingly pure notion of computation based only on the notion of function, that is, Church's λ-calculus [CR36]. There are several reasons to do so.

- We will see a number of important concepts in their simplest possible form, which means we can discuss them in full detail. We will then reuse these notions frequently throughout the course without the same level of detail.

- The λ-calculus is of great historical and foundational significance. The independent and nearly simultaneous development of Turing Machines [Tur36] and the λ-Calculus [CR36] as universal computational mechanisms led to the *Church-Turing Thesis*, which states that the effectively computable (partial) functions are exactly those that can be implemented by Turing Machines or, equivalently, in the λ-calculus.

- The notion of function is the most basic abstraction present in nearly all programming languages. If we are to study programming languages, we therefore must strive to understand the notion of function.

- It's cool!

In mathematical practice, functions are ubiquitous. For example, we

might define

$$\begin{array}{lcl} f(x) & = & x + 5 \\ g(x, y) & = & 2x + y \end{array}$$

Oddly, we never state what f or g actually are, we only state what happens when we apply them to arbitrary arguments such as x or y. The λ-calculus starts with the simple idea that we should have notation for the function itself, the so-called λ-abstraction.

$$\begin{array}{l} f = \lambda x.\, x + 5 \\ g = \lambda x. \lambda y.\, 2x + y \end{array}$$

Syntax We can already see that in a pure calculus of functions we will need at least three different kinds of expressions that we define inductively as follows.

- *Variables* x, y, z, etc. are expressions.
- If e is an expression then the λ*-abstractions* $\lambda x.\, e$ is an expression.
- If e_1 and e_2 are expressions then the *application* $e_1\, e_2$ is an expression.

We summarize this inductive definition in the following form.

$$\begin{array}{lcl} \text{Variables} & & x, y, z, \ldots \\ \text{Expressions} & e \quad ::= & \lambda x.\, e \mid e_1\, e_2 \mid x \end{array}$$

This is the definition of the *concrete syntax* of the λ-calculus that we use in the implementation together with additional conventions and notations such as parentheses to avoid ambiguity.

1. Juxtaposition (which expresses application) is *left-associative* so that $x\, y\, z$ is read as $(x\, y)\, z$.
2. $\lambda x.$ is a prefix whose scope extends as far as possible while remaining consistent with the parentheses that are present. For example, $\lambda x.\, (\lambda y.\, x\, y\, z)\, x$ is read as $\lambda x.\, ((\lambda y.\, (x\, y)\, z)\, x)$.

Often, we also define the *abstract syntax* of a language that is based on *abstract binding trees* (ABTs), which are a generalization of abstract syntax trees.

$$\begin{array}{lcll} & & \text{Concrete} & \text{Abstract} \\ \text{Expressions} \quad e \quad ::= & \lambda x.\, e & \mathsf{lam}(x.e) \\ & & e_1\, e_2 & \mathsf{app}(e_1, e_2) \end{array}$$

In an ABT, we have different operators that indicate the binding of variables. In the λ-calculus, there are two operators, one for lambda abstraction and one for function application. A variable is always an ABT and is not explicitly mentioned. We will revisit this concept later. So the identity function has the abstract syntax $\mathsf{lam}(x.x)$ and concrete syntax $\lambda x.\, x$.

Semantics The meaning or semantics of an expression is linked to the meaning of a variable. Here, a variable is a placeholder for an expression. We could therefore call variables also *expression variables*. That means that if we have an expression e that contains a variable x then we can replace x with an expression e' to obtain another expression. We say that e' is substituted for x in e and write $[e'/x]e$. For now, we will use this notion of substitution informally—in the next lecture we will define it formally.

The abstraction $\lambda x.\, e$ for some arbitrary expression e stands for the function, which, when applied to some e' becomes $[e'/x]e$, that is, the result of substituting e' for all (free) occurrences of the variable x in e.

We say $\lambda x.\, e$ *binds* the variable x with scope e. Variables that occur in e but are not bound are called *free variables*, and we say that a variable x may occur free in an expression e. For example, y is free in $\lambda x.\, x\, y$ but not x. Bound variables can be renamed consistently in an expression. So we consider $\lambda x.\, x + 5$ to be equal to $\lambda y.\, y + 5$ and $\lambda whatever.\, whatever + 5$. Generally, we rename variables *silently* because we identify expressions that differ only in the names of λ-bound variables. But, if we want to make the step explicit, we call it α-conversion.

$$\lambda x.\, e =_\alpha \lambda y.[y/x]e \quad \text{provided } y \text{ not free in } e$$

The proviso is necessary, for example, because $\lambda x.x\, y \neq \lambda y.y\, y$.

We capture the rule for function application with

$$(\lambda x.\, e_2)\, e_1 =_\beta [e_1/x]e_2$$

and call it β*-conversion*. Some care has to be taken for the substitution to be carried our correctly—we will return to this point later.

If we think beyond mere equality at *computation*, we see that β-conversion has a definitive direction: we apply is from left to right. We call this β-*reduction* and it is the engine of computation in the λ-calculus.

$$(\lambda x.\, e_2)\, e_1 \longmapsto_\beta [e_1/x]e_2$$

3 Simple Functions and Combinators

The simplest functions are the identity function and the constant function. The identity function, called *I*, just returns its argument x.

$$I \triangleq \lambda x.\, x$$

The constant function returning x could be written as

$$\lambda y.\, x$$

We calculate

$$(\lambda y.\, x)\, e \longmapsto_\beta x$$

for any expression e since y does not occur in the expression x. This is somewhat incomplete in the sense the expression $\lambda y.\, x$ has a *free variable*. What we would like is an expression *K* without free variables, such $K\, x$ is the constant function, always returning x. But that's easy: we just abstract over x.

$$K \triangleq \lambda x.\, \lambda y.\, x$$

Then $K\, x \longmapsto_\beta \lambda y.\, x$ is the constant function returning x.

We call expressions without free variable *closed expression*. A *combinator* is just a closed λ-expression like *I* or *K*. We will see more interesting combinators in the next lecture.

4 Function Composition

One the most fundamental operation on functions in mathematics is to compose them. We might write

$$(f \circ g)(x) = f(g(x))$$

Having λ-notation we can first explicitly denote the result of composition (with some redundant parentheses)

$$f \circ g = \lambda x.\, f(g(x))$$

As a second step, we realize that $\circ$ itself is a function, taking two functions as arguments and returning another function. Ignoring the fact that it is usually written in infix notation, we define

$$\circ \triangleq B \triangleq \lambda f.\, \lambda g.\, \lambda x.\, f\, (g\, x)$$

We call it B because that's its traditional name as a combinator.

One of the fundamental properties of function composition is that it is associative, that is, $(f \circ g) \circ h = f \circ (g \circ h)$. If our representation of function composition is correct, we should be able to verify

$$B\, f\, (B\, g\, h) = B\, (B\, f\, g)\, h$$

In general, two expressions e_1 and e_2 are equal (or β-equivalent) if we can transform e_1 into e_2 using a series of β- and α-conversions. Verify that this relation is reflexive, symmetric, and transitive.

Using this definition, we aim to verify the aforementioned equality. Let's start with the left side and the right side and apply β-reduction. The definition of B takes place here in our language of mathematical discourse, to replacing its definition is not actually a step of β-equality but just equality. We highlight in red the variable or binder that is being replaced, renamed, or substituted for in the following step.

$$\begin{array}{ll}
 & B\, f\, (B\, g\, h) \\
= & (\lambda f.\, \lambda g.\, \lambda x.\, f\, (g\, x))\, f\, (B\, g\, h) \\
=_\beta & (\lambda g.\, \lambda x.\, f\, (g\, x))\, (B\, g\, h) \\
=_\beta & \lambda x.\, f\, ((B\, g\, h)\, x) \\
= & \lambda x.\, f\, (((\lambda f.\, \lambda g.\, \lambda x.\, f\, (g\, x))\, g\, h)\, x) \\
=_\alpha & \lambda x.\, f\, (((\lambda f.\, \lambda g'.\, \lambda x.\, f\, (g'\, x))\, g\, h)\, x) \\
=_\beta & \lambda x.\, f\, ((\lambda g'.\, \lambda x.\, g\, (g'\, x))\, h\, x) \\
=_\beta & \lambda x.\, f\, ((\lambda x.\, g\, (h\, x))\, x) \\
=_\beta & \lambda x.\, f\, (g\, (h\, x))
\end{array}$$

Note that the renaming from g to some other variable name g' (α-conversion) is necessary, because otherwise the variable g would be *captured* by the binder on g, giving us the wrong answer:

$$\begin{array}{ll}
 & \lambda x.\, f\, (((\lambda f.\, \lambda g.\, \lambda x.\, f\, (g\, x))\, g\, h)\, x) \\
\neq_\beta & \lambda x.\, f\, ((\lambda g.\, \lambda x.\, g\, (g\, x))\, h\, x) \\
=_\beta & \lambda x.\, f\, (h\, (h\, x))
\end{array}$$

This is one of the last times we'll be explicit about α-conversion and we'll just silently apply it as needed.

With a similar chain of reasoning we can verify that for the right-hand side we have

$$B\, (B\, f\, g)\, h =_\beta \lambda x.\, f\, (g\, (h\, x))$$

Therefore, by transitivity and symmetry of equality, we know that function composition is associative as it should be.

References

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[Tur36] Alan Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936. Published 1937.