Lecture Notes on The Lambda Calculus

15-814: Types and Programming Languages
Frank Pfenning

Lecture 1
Tuesday, September 3, 2019

1 Introduction

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!

2 The λ-Calculus

In ordinary mathematical practice, functions are ubiquitous. For example, we might define

$$\begin{aligned} f(x) &= x + 5 \\ g(y) &= 2 * y + 7 \end{aligned}$$

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.

$$\begin{aligned} f &= \lambda x.\, x + 5 \\ g &= \lambda y.\, 2 * y + 7 \end{aligned}$$

In general, $\lambda x.\, e$ 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* or *plugging in* e' for occurrences of the variable x in e. For now, we will use this notion of substitution informally—in the next lecture we will define it formally.

We can already see that in a pure calculus of functions we will need at least three different kinds of expressions: λ-abstractions $\lambda x.\, e$ to form function, *application* $e_1\, e_2$ to apply a function e_1 to an argument e_2, and *variables* x, y, z, etc. We summarize this in the following form

$$\begin{array}{llcl} \text{Variables} & x & & \\ \text{Expressions} & e & ::= & \lambda x.\, e \mid e_1\, e_2 \mid x \end{array}$$

This is not the definition of the *concrete syntax* of a programming language, but a slightly more abstract form called *abstract syntax*. When we write down concrete expressions there are 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 $\lambda x.\, ((\lambda y.\, (x\, y)\, z)\, x)$.

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 a term So $\lambda x.\, x + 5 = \lambda y.\, y + 5 = \lambda whatever.\, whatever + 5$. Generally, we rename variables *silently* because we identify terms 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.

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

3 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

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

Now we can calculate, for example, the composition of the two functions we had at the beginning of the lecture. We note the steps where we apply

β-conversion.

$$\begin{array}{ll} & (\circ\,(\lambda x.\,x+5))\,(\lambda y.\,2*y+7) \\ = & ((\lambda f.\,\lambda g.\,\lambda x.\,f(g(x)))(\lambda x.\,x+5))\,(\lambda y.\,2*y+7) \\ =_\beta & (\lambda g.\,\lambda x.\,(\lambda x.\,x+5)(g(x)))\,(\lambda y.\,2*y+7) \\ =_\beta & \lambda x.\,(\lambda x.\,x+5)\,((\lambda y.\,2*y+7)(x)) \\ =_\beta & \lambda x.\,(\lambda x.\,x+5)\,(2*x+7) \\ =_\beta & \lambda x.\,(2*x+7)+5 \\ = & \lambda x.\,2*x+12 \end{array}$$

While this appears to go beyond the pure λ-calculus, we will see in Section 6 that we can actually encode natural numbers, addition, and multiplication. We can see that $\circ$ as an operator is not *commutative* because, in general, $\circ\, f\, g \neq \circ\, g\, f$. You may test your understanding by calculating $(\circ\,(\lambda y.\,2*y+7))\,(\lambda x.\,x+5)$ and observing that it is different.

4 Summary of λ-Calculus

λ-Expressions.

Variables	x		
Expressions	e	$::=$	$\lambda x.\,e \mid e_1\,e_2 \mid x$

$\lambda x.\,e$ binds x with scope e, which is as large as possible while remaining consistent with the given parentheses. Juxtaposition $e_1\,e_2$ is left-associative.

Equality.

Substitution	$[e_1/x]e_2$			*(capture-avoiding, see Lecture 2)*
α-conversion	$\lambda x.\,e$	$=_\alpha$	$\lambda y.[y/x]e$	provided y not free in e
β-conversion	$(\lambda x.\,e_2)\,e_1$	$=_\beta$	$[e_1/x]e_2$	

We generally apply α-conversion silently, identifying terms that differ only in the names of the bound variables.

Reduction.

β-reduction	$(\lambda x.\,e_2)\,e_1$	$\longrightarrow_\beta$	$[e_1/x]e_2$

5 Representing Booleans

Before we can claim the λ-calculus as a universal language for computation, we need to be able to represent *data*. The simplest nontrivial data type are the Booleans, a type with two elements: *true* and *false*. The general technique is to represent the values of a given type by *normal forms*, that is, expressions that cannot be reduced. Furthermore, they should be *closed*, that is, not contain any free variables. We need to be able to distinguish between two values, and in a closed expression that suggest introducing two bound variables. We then define rather arbitrarily one to be *true* and the other to be *false*

$$\begin{array}{rcl} \mathit{true} & = & \lambda x.\, \lambda y.\, x \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y \end{array}$$

The next step will be to define *functions* on values of the type. Let's start with negation: we are trying to define a λ-expression *not* such that

$$\begin{array}{rcl} \mathit{not}\ \mathit{true} & =_\beta & \mathit{false} \\ \mathit{not}\ \mathit{false} & =_\beta & \mathit{true} \end{array}$$

We start with the obvious:

$$\mathit{not} = \lambda b.\ \ldots$$

Now there are two possibilities: we could either try to apply b to some arguments, or we could build some λ-abstractions. In lecture, we followed both paths. Let's first try the one where b is applied to some arguments.

$$\mathit{not} = \lambda b.\, b\,(\ldots)\,(\ldots)$$

We suggest two arguments to b, because b stands for a Boolean, and Booleans *true* and *false* both take two arguments. $\mathit{true} = \lambda x.\, \lambda y.\, x$ will pick out the first of these two arguments and discard the second, so since we specified $\mathit{not}\ \mathit{true} = \mathit{false}$, the first argument to b should be *false*!

$$\mathit{not} = \lambda b.\, b\ \mathit{false}\ (\ldots)$$

Since $\mathit{false} = \lambda x.\, \lambda y.\, y$ picks out the second argument and $\mathit{not}\ \mathit{false} = \mathit{true}$, the second argument to b should be *true*.

$$\mathit{not} = \lambda b.\, b\ \mathit{false}\ \mathit{true}$$

Now it is a simple matter to calculate that the computation of *not* applied to *true* or *false* completes in three steps and obtain the correct result.

$$\begin{array}{rcl} \mathit{not}\ \mathit{true} & \longrightarrow_\beta^3 & \mathit{false} \\ \mathit{not}\ \mathit{false} & \longrightarrow_\beta^3 & \mathit{true} \end{array}$$

We write $\longrightarrow_\beta^n$ for reduction in n steps, and $\longrightarrow_\beta^*$ for reduction in an arbitrary number of steps, including zero steps. In other words, $\longrightarrow_\beta^*$ is the reflexive and transitive closure of $\longrightarrow_\beta$.

An alternative solution hinted at above is to start with

$$\mathit{not}' = \lambda b.\, \lambda x.\, \lambda y.\, \ldots$$

We pose this because the result of *not* b should be a Boolean, and the two Booleans both start with two λ-abstractions. Now we reuse the previous idea, but apply b not to *false* and *true*, but to y and x.

$$\mathit{not}' = \lambda b.\, \lambda x.\, \lambda y.\, b\, y\, x$$

Again, we calculate

$$\begin{array}{lcl} \mathit{not}'\ \mathit{true} & \longrightarrow_\beta^3 & \mathit{false} \\ \mathit{not}'\ \mathit{false} & \longrightarrow_\beta^3 & \mathit{true} \end{array}$$

An important observation here is that

$$\mathit{not} = \lambda b.\, b\ (\lambda x.\, \lambda y.\, y)\ (\lambda x.\, \lambda y.\, x) \neq \lambda b.\, \lambda x.\, \lambda y.\, b\, y\, x = \mathit{not}'$$

Both of these are *normal forms* (they cannot be reduced) and therefore represent *values* (the results of computation). Both correctly implement negation on Booleans, but they are *different*. This is evidence that when computing with particular data representations in the λ-calculus it is *not extensional*: even though the functions behave the same on all the arguments we care about (here just *true* and *false*), the are not convertible. To actually see that they are not convertible we need the Church-Rosser theorem which says if e_1 and e_2 are $\alpha\beta$-convertible then there is a common reduct e such that $e_1 \longrightarrow_\beta^* e$ and $e_2 \longrightarrow_\beta^* e$.

As a next exercise we try exclusive or. We want to define a λ-expression *xor* such that

$$\begin{array}{lcl} \mathit{xor}\ \mathit{true}\ \mathit{true} & =_\beta & \mathit{false} \\ \mathit{xor}\ \mathit{true}\ \mathit{false} & =_\beta & \mathit{true} \\ \mathit{xor}\ \mathit{false}\ \mathit{true} & =_\beta & \mathit{true} \\ \mathit{xor}\ \mathit{false}\ \mathit{false} & =_\beta & \mathit{false} \end{array}$$

Learning from the negation, we start by guessing

$$\mathit{xor} = \lambda b.\, \lambda c.\, b\, (\ldots)\, (\ldots)$$

where we arbitrarily put b first. Looking at the equations, we see that if b is *true* then the result is always negation of c.

$$\mathit{xor} = \lambda b.\, \lambda c.\, b\, (\mathit{not}\ c)\, (\ldots)$$

If b is *false* the result is always just c, no matter what c is.

$$\mathit{xor} = \lambda b.\, \lambda c.\, b\,(\mathit{not}\ c)\, c$$

Again, it is now a simple matter to verify the desired equations and that, in fact, the right-hand side of these equations is obtained by reduction.

6 Representing Natural Numbers

Finite types such as Booleans are not particularly interesting. When we think about the computational power of a calculus we generally consider the *natural numbers* $0, 1, 2, \ldots$. We would like a representation $\overline{n}$ such that they are all distinct. We obtain this by thinking of the natural numbers are generated from zero by repeated application of the successor function. Since we want our representations to be closed we start with two abstractions: one (z) that stands for zero, and one (s) that stands for the successor function.

$$\begin{array}{lcl}
\overline{0} & = & \lambda s.\, \lambda z.\, z \\
\overline{1} & = & \lambda s.\, \lambda z.\, s\, z \\
\overline{2} & = & \lambda s.\, \lambda z.\, s\,(s\, z) \\
\overline{3} & = & \lambda s.\, \lambda z.\, s\,(s\,(s\, z)) \\
\ldots & & \\
\overline{n} & = & \lambda s.\, \lambda z.\, \underbrace{s\,(\ldots(s}_{n \text{ times}}\ z))
\end{array}$$

In other words, the representation $\overline{n}$ iterates its first argument n times over its second argument

$$\overline{n}\, f\, x = f^n(x)$$

where $f^n(x) = \underbrace{f(\ldots(f}_{n \text{ times}}(x)))$

The first order of business now is to define a successor function that satisfies $\mathit{succ}\ \overline{n} = \overline{n+1}$. As usual, there is more than one way to define it, here is one (throwing in the definition of *zero* for uniformity):

$$\begin{array}{lclcl}
\mathit{zero} & = & \overline{0} & = & \lambda s.\, \lambda z.\, z \\
\mathit{succ} & = & \lambda n.\, \overline{n+1} & = & \lambda n.\, \lambda s.\, \lambda z.\, s\,(n\, s\, z)
\end{array}$$

We cannot carry out the correctness proof in closed form as we did for the Booleans since there would be infinitely many cases to consider. Instead we calculate generically (using mathmetical notation and properties)

$$\begin{array}{rl} & \mathit{succ}\ \overline{n} \\ = & \lambda s.\, \lambda z.\, s\, (\overline{n}\, z\, s) \\ = & \lambda s.\, \lambda z.\, s\, (s^n(z)) \\ = & \lambda s.\, \lambda z.\, s^{n+1}(z) \\ = & \overline{n+1} \end{array}$$

A more formal argument might use mathematical induction over n.

Using the iteration property we can now define other mathematical functions over the natural numbers. For example, addition of n and k iterates the successor function n times on k.

$$\mathit{plus} = \lambda n.\, \lambda k.\, n\ \mathit{succ}\ k$$

You are invited to verify the correctness of this definition by calculation. Similarly:

$$\begin{array}{lcl} \mathit{times} & = & \lambda n.\, \lambda k.\, n\ (\mathit{plus}\ k)\ \mathit{zero} \\ \mathit{exp} & = & \lambda b.\, \lambda e.\, e\ (\mathit{times}\ b)\ (\mathit{succ}\ \mathit{zero}) \end{array}$$

More about this and other properties and examples of the λ-calculus in Lecture 2.

7 Exercises

Exercise 1 *Define the following functions on Booleans in at least two distinct ways.*

1. *Conjunction "and".*
2. *The conditional "if" such that*

$$\begin{array}{lcl} \mathit{if}\ \mathit{true}\ e_1\ e_2 & =_\beta & e_1 \\ \mathit{if}\ \mathit{false}\ e_1\ e_2 & =_\beta & e_2 \end{array}$$

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.