Lecture Notes on
The Lambda Calculus

15-814: Types and Programming Languages
Frank Pfenning

Lecture 1
Tuesday, August 31, 2021

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

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

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.

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

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

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

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 $\lambda x.\, x + 5 = \lambda y.\, y + 5 = \lambda \mathit{whatever}.\, \mathit{whatever} + 5$. 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 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.

$$I = \lambda x.\, x$$

The constant function returning x could be written as

$$\lambda y.\, x$$

We calculate

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

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* which is therefore fixed. What we would like is a *closed expression K* (one without free variables) such $K\, x$ is the constant function, always returning x. But that's easy: we just abstract over x!

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

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

A combinator for us 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

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

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

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

We hope we can do this by pure calculation, so 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, but transitivity and symmetry of equality we know that function composition is associative as it should be.

5 Summary of λ-Calculus

λ-Expressions.

Variables	$x, y, z, \ldots$		
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. Because β-conversion is a form of equality it is reflexive, symmetric, and transitive and can be applied anywhere in an expression.

Reduction.

$$\beta\text{-reduction} \quad (\lambda x.\, e_2)\, e_1 \quad \longrightarrow_\beta \quad [e_1/x]e_2$$

Reduction is oriented, so we don't think of it as an equality and it is neither reflexive, nor symmetric, nor transitive. On the other hand it is a congruence, that is, we can apply β-reduction anywhere in an expression.

6 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 conjunction. We want to define a λ-expression *and* such that

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

Learning from the negation, we start by guessing

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

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

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

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

$$\mathit{and} = \lambda b.\, \lambda c.\, b\, c\, \mathit{false}$$

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.

7 Nontermination

At this point we pause briefly to ask three natural questions:

1. Does every expression have a normal form?
2. Can we always compute a normal form if one exists?
3. Are normal forms unique?

The answers to these questions are crucial to understanding to what extent we might consider the λ-calculus a universal model of computation.

Does every expression have a normal form?

If the λ-calculus is to be equivalent in computational power to Turing machines in some way, then we would expect the answer to be "no" because computations of Turing machines may not halt. However, it is not immediate to think of some expression that doesn't have a normal form. If you haven't seen something like this already, you may want to play around with some expressions to see if you can come up with one.

The simplest one is

$$\Omega = (\lambda x.\, x\, x)\, (\lambda x.\, x\, x)$$

Indeed, there is only one possible β-reduction and it immediately leads to exactly the same term:

$$\begin{array}{rcl} \Omega & = & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longrightarrow_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longrightarrow_\beta & (\lambda x.\, x\, x)\, (\lambda x.\, x\, x) \\ & \longrightarrow_\beta & \ldots \end{array}$$

So Ω reduces in one step to itself and only to itself.

Can we always compute a normal form if one exists?

The answer here is "yes", although it is not easy to prove that this is the case. Let's consider an example (recall that $K = \lambda x.\, \lambda y.\, x$):

$$K\, I\, \Omega \longrightarrow_\beta (\lambda y.\, I)\, \Omega \longrightarrow_\beta I$$

So the expression $K\, I\, \Omega$ does have a normal form, even though Ω does not. This is because the constant function $K\, I$ ignores its argument. On the other hand we also have

$$K\, I\, \Omega \longrightarrow_\beta K\, I\, \Omega \longrightarrow_\beta K\, I\, \Omega \longrightarrow_\beta \cdots$$

because we have the $\Omega \longrightarrow_\beta \Omega$ and reduction can be applied anywhere in an expression.

Fortunately, there is a strategy which turns out to be complete in the sense that if an expression has a normal form, this strategy will find it. It is called *leftmost-outermost* or *normal-order reduction*. This strategy scans through the expression from left to right and when it find a *redex* (that is, an expression of the form $(\lambda x.\, e)\, e'$) it applies β-reduction and then returns to the beginning of the result expression. In particular, it does not consider any redex in e or e', only the "outermost" one. Also, in an expression $((\lambda x.\, e_1)\, e_2)\, e_3$ it does not consider any potential redex in e_3, only the leftmost one.

This strategy works in our example: the redex in Ω would not be considered, only the redex $K\, I$ and then the redex $(\lambda y.\, I)\, \Omega$.

The implementation of LAMBDA uses a straightforward function for leftmost-outermost reduction, complicated very slightly by the fact that names such as K or I which in the notes are only abbreviations at the mathematical level of discourse, are actual language-level definitions in the implementation. So we have to expand the definition of K, for example, before applying β-reduction, but we do not officially count this as a substitution.

The notion of leftmost-outermost reduction is closely related to the notion of call-by-name evaluation in programming languages (and, with a little more distance, to call-by-need which is employed in Haskell). In contrast, call-by-value would reduce the argument of a function before applying the β-reduction, which is not complete, as our example shows. The analogy is not exact, however, since in programming languages such as ML or Haskell we also do not reduce under λ-abstractions, a fact that represents a sharp dividing line between foundational calculi such as the λ-calculus and actual programming languages. We will justify and understand these decisions in a few lectures.

Are normal forms unique?

The outcome of a computation starting from e is its normal form. At any point during a computation there may be many redices. Ideally, the outcome would be independent of the reduction strategy we choose as long as we reach a normal form. Otherwise, the meaning of an expression (as represented by its normal form) may be ambiguous. Therefore, Church and Rosser [CR36] spend considerable effort in proving the uniqueness of normal forms. The key technical device is a property called *confluence* (also referred to as the *Church-Rosser property*). It is often depicted in the following diagram:

$$\begin{tikzcd}
 & e \arrow[dl, "*"'] \arrow[dr, "*"] & \\
e_1 \arrow[dr, dashed, "*"'] & & e_2 \arrow[dl, dashed, "*"] \\
 & e' &
\end{tikzcd}$$

In words: if we can reduce e to e_1 and also e to e_2 then there exists an e' such that e_1 and e_2 both reduce to e'. The solid lines are given reduction sequences while the reduction sequences represented by dashed lines have to be shown to exist. Reduction here is in multiple steps (indicated by the star "$*$"). For the λ-calculus (and the original Church-Rosser Theorem), this reduction would usually be β-reduction. Very roughly, the proof shows how to simulate the steps from e to e_2 when starting from e_1 and (symmetrically) simulate the steps from e to e_1 when starting from e_2.

Confluence implies the uniqueness of normal forms. Suppose e_1 and e_2 in the diagram are normal forms. Because they cannot be reduced further, the sequence of reductions to e' must consist of zero steps, so $e_1 = e' = e_2$.

Confluence implies that even though we might embark on an unfortunate path (for example, keep reducing Ω in $K\,I\,\Omega$) we can still recover if indeed there is a normal form. In this example, we might eventually decide to reduce $K\,I$ and then the redex $(\lambda y.\, I)\,\Omega$.

8 The LAMBDA Language

In lecture, we used a toy implementation of a the λ-calculus in a language called LAMBDA. This implementation uses a *concrete syntax* where λ is written as a backslash '\'. A program consists of a sequence of declarations, of which there are three forms:

defn $x = e$	variable x stands for e
norm $x = e$	variable x stands for the normal form of e
conv $e_1 = e_2$	verify that e_1 and e_2 have the same normal form

Allowing definitions is a convenience, but it does not change the expressive power of the λ-calculus, because we can replace **defn** $x = e$ by $(\lambda x.\, \ldots)\, e$ where '...' represents the scope of the definition. The **norm** and **conv** declarations initiate computation and allow the programmer to examine the normal form of an expression (if it exists).

In addition, declarations can be negated with **fail**, for example, to check that two expressions are not convertible.

```
% represent booleans as closed expressions in normal form

defn true = \x. \y. x
defn false = \x. \y. y

defn not = \b. b false true
defn not' = \b. \x. \y. b y x

(* confirm that not and not' are not convertible *)
fail conv not = not'

% normalize "not true"
norm _ = not true
```

```

% test not and not' against their specification
conv not true = false
conv not false = true

conv not' true = false
conv not' false = true
```

Listing 1: Booleans in LAMBDA

For more information on LAMBDA, consult the Software page for the course. The live code from lecture and a trimmed down version can be found at `01-lambda/`.

Exercises

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

1. "*nor*", the negation of disjunction
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.