Theory of Formal Languages (Introduction to Lambda-calculus)

Lecture 16: April 27, 2021

Prof. K.R. Chowdhary *: Professor of CS*

16.1 Introduction

In the 1930s, several people were interested in the question: what does it mean for a function $f : \mathbb{N} \to \mathbb{N}$ to be computable? An informal definition of computability is that there should be a *pencil-and-paper* method allowing a trained person to calculate $f(n)$, for any given n. The concept of a pencil-and-paper method is not so easy to formalize. Three different researchers attempted to do so, resulting in the following definitions of computability:

1. Turing defined an idealized computer we now call a *Turing machine*, and postulated that a function is computable (in the intuitive sense) if and only if it can be computed by such a machine.

2. *Gödel* defined the class of *general recursive functions* as the smallest set of functions containing all the constant functions, the successor function, and closed under certain operations (such as compositions and recursion). He postulated that a function is computable (in the intuitive sense) if and only if it is general recursive.

3. Church defined an idealized programming language called the *lambda calculus*, and postulated that a function is computable (in the intuitive sense) *if and only if* it can be written as a lambda term.

It was proved by Church, Kleene, and Turing that all three computational models were equivalent to each other, i.e., each model defines the same class of computable functions. Whether or not they are equivalent to the "intuitive" notion of computability is a question that cannot be answered, because there is no formal definition of "intuitive computability". The assertion that they are in fact equivalent to intuitive computability is known as the Church-Turing thesis.

Since the meaning of a computer program can be considered as a function from input values to output values, the functions play a prominent role in describing the semantics of a programming language Alonzo Church developed the lambda calculus (also written as λ-calculus) in the 1930s as a theory of functions that provides rules for manipulating functions in a purely *syntactic* manner. The λ-calculus is the simplest programming language, it consists of single transformation rule (variable substitution, also called β-conversion) and a function definition scheme. Although the lambda calculus arose as a branch of mathematical logic to provide a foundation for mathematics, it has led to considerable ramifications in the theory of programming languages, and has been used to model the programming languages [kjean07].

The λ-calculus is universal programming language in the sense that any computable function can be expressed and evaluated using this formalism, it is thus equivalent to Turing machines. However, the λ-calculus emphasizes the use of symbolic transformation rules and does not care about the actual machine implementation. Thus, it is an approach more related to software than hardware.

Lambda calculus is a formal system in mathematical logic for expressing computation based on function *abstraction* and *application* using variable binding and substitution. It is a universal model of computation that can be used to simulate any Turing machine.

Some of the features of λ-calculus are as follows:

- The λ-calculus can be used to encode programs and data, such as Booleans and natural numbers,
- It is the simplest possible programming language, that is Turing complete,
- 'Pure LISP' is equivalent Lambda Calculus, and
- 'LISP' is Lambda calculus, plus some additional features such as data types, input/output, etc.

Beyond the influence of the lambda calculus in the area of computation theory, it has contributed important results to the formal semantics of programming languages in the following ways:

- The lambda calculus has the power to represent all computable functions, its uncomplicated syntax and semantics provide an excellent vehicle for studying the meaning of programming language concepts.
- All functional programming languages can be viewed as syntactic variations of the lambda calculus, so that both their semantics and implementation can be analyzed in the context of the lambda calculus.
- *Denotational semantics*, one of the foremost methods of formal specification of languages, grew out of research in the lambda calculus and expresses its definitions using the higher-order functions of the lambda calculus.

16.2 Functional Programming languages

In a nutshell, the lambda calculus can be described as a general theory of computable functions since it provides a formalism for dealing with *functions-as-terms* in the context of foundations of mathematics. In axiomatic set theories, functions from a given set to itself are defined by means of their graphs; that is, as – typically infinite – set of ordered pairs. By contrast, in lambda calculus functions are implemented by means of some – always finite – formal expressions called *terms*. Furthermore, in any ordinary set theory it follows from the axioms that a function cannot be a member of its domain. In lambda calculus, on the other hand, every term is in the domain of a given function and in particular every function can take itself as input.

Apart from these early foundational issues, the most interesting feature of this calculus – as well as the real motivation for its success – is that it constitutes a *model of computation* and for this reason it is the theoretical core of many modern functional programming languages [michelb21].

A mathematical function is a mapping of domain set to range set. A function definition is the description of this mapping either explicitly by enumeration or implicitly by an expression. In conventional definition of a function is specified by a function name followed by a list of parameters in parenthesis, followed by the expression describing the mapping, e.g., $cube(x) \equiv x*x*x$, where x is a real number. Alonzo Church introduced the notation of nameless functions using the Lambda notation. A lambda expression specifies the parameters and the mapping of a function using the λ operator, e.g., $\lambda(x)\ x*x*x$. It is the function itself, so the notation of applying the example nameless function to a certain argument is, for example, $(\lambda(x)\ x*x*x)(4)$.

Programming in a functional language consists of building function definitions and using the computer to evaluate expressions, i.e. function application with concrete arguments. The major programming task is then to construct a function for a specific problem by combining previously defined functions according to mathematical principles. The main task of the computer is to evaluate function calls and to print the resulting function values. This way the computer is used like an ordinary pocket computer, of course at a much more flexible and powerful level.

A characteristic feature of functional programming is that if an expression possesses a well-defined value, then the order in which the computer performs the evaluation does not affect the result of the evaluation. Thus, the result of the evaluation of an expression is just its value. This means that in a pure functional language no side-effects exist. Side-effects are connected to variables that model memory locations. Thus, in a pure functional programming language no variables exists in the sense of imperative languages. The major control-flow methods are *recursion* and *conditional expressions*. This is quite different from imperative languages, in which the basic means for control are *sequencing* and *iteration*. Functional programming also supports the specification of higher-order functions. A higher-order function is a function definition which allows functions as arguments or returns a function as its value.

All these aspects together, but especially the latter are major sources of the benefits of functional programming style in contrast to imperative programming style, viz. that functional programming provides a high-level degree of modularity. When defining a problem by dividing it into a set of sub-problems, a major issue concerns the ways in which one can glue the (sub-) solutions together. Therefore, to increase ones ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language – a major strength of functional programming.

The lambda calculus is an idealized programming language; arguably, it is the simplest possible programming language that is Turing complete. Because of its simplicity, it is a useful tool for defining and proving properties of programs. Many real-world programming languages can be regarded as extensions of the lambda-calculus. This is true for all functional programming languages, a class that includes *Lisp*, *Scheme*, *Haskell*, and *ML*(Meta Language). These languages combine the lambda calculus with additional features, such as data types, input/output, side effects, udpateable memory, object orientated features, etc. The lambda calculus provides a vehicle for studying such extensions, in isolation and jointly, to see how they will affect each other, and to prove properties of programming language (such as: a well-formed program will not crash). The lambda calculus is also a tool used in compiler construction.

A functional program consists of an expression E (representing both the algorithm and the input). This expression E is subject to some rewrite rules. Reduction consists of replacing a part P of E by another expression P' according to the given rewrite rules. In schematic notation,

$$E[P] \to E[P'], \tag{16.1}$$

provided that $P \to P'$ is according to the rules. This process of reduction will be repeated until the resulting expression has no more parts that can be rewritten. This so called *normal form* E^* of the expression E, and consists of the output of the given functional program. An example is,

$$\begin{aligned}(7+4)*(8+5*3) &\to 11*(8+5*3)\\ &\to 11*(8+15)\\ &\to 11*23\\ &\to 253.\end{aligned}$$

In this example, the reduction rules are the "tables" of addition and of multiplication on the numerals. Reduction systems usually satisfy the *Church-Rosser* property, which states that the normal form obtained is independent of the order of evaluation of subterms. Indeed, the first example above may be reduced in another style as follows.

$$\begin{aligned}(7+4)*(8+5*3) &\to (7+4)*(8+15)\\ &\to (7+4)*23\\ &\to 11*23\\ &\to 253.\end{aligned}$$

16.3 A quick introduction to λ-calculus

The lambda calculus is a theory of functions as formulas. It is a system for manipulating functions as expressions. Let us begin by looking at another well-known language of expressions, namely arithmetic, these are made up from variables $(a,b,...,x,y,z,x_0,x_1,\dots)$, numbers $(1,2,3,...)$, and operators $\{+,-,\times\}$ etc. An expression such as $x+y$ stands for the result of an addition (as opposed to an instruction to add or the statement that something is being added).

We define the syntax of λ-calculus in simple way, as follows:

- x is treated as variable, which can be a character or string representing a parameter or mathematical/logical value,
- $(\lambda x.M)$ is called *abstraction*. It is function definition (M is a lambda term), the variable x becomes bound in the expression,
- $(M\ \ N)$ is called *application*, where a function is applied to an argument, the M and N are called lambda terms.

Definition 16.1 *An* abstraction *is inverse operation from reduction. If $A > B$, we say A comes from B via abstraction. We may also write $B < A$.*

The great advantage of this language is that expressions can be nested without any need to mention the intermediate results explicitly. So for instance, we write

$$A = (x+y) \times z^2,$$

and not the following: let $w = x+y$, then let $u = z^2$, then let $A = w \times u$. The latter notation would be tiring and cumbersome to manipulate. The *lambda calculus* extends the idea of an expression language to include functions, where we normally write,

$$A = (\lambda x.x^2)(5).$$

The expression $\lambda x.x^2$ stands for the function that maps x to x^2 (as opposed to the statement that x is being mapped to x^2). As in arithmetic, we use parentheses to group terms. It is understood that the variable x is a *local variable* in the term $\lambda x.x^2$. Thus, it does not make any difference if we write $\lambda y.y^2$ instead. A local variable is also called a *bound* variable.

The set of lambda terms is built up from variables using *abstraction*

$$(\lambda x.M), \tag{16.2}$$

and *application*

$$(M\ \ N). \tag{16.3}$$

In 16.2 above, x is any variable and M, N are lambda terms. $(\lambda x.M)$ is the function that maps x to M. We sometimes omit parenthesis, understanding abstraction to associate to right, and application to associate to left. For example, $\lambda x.\lambda y.x\ y\ x$ denotes $(\lambda x.(\lambda y.((x\ y)x)))$. We can also join the consecutive abstractions as in $\lambda x\ y.x\ y\ x$. The λ term binds the variable x in the body term M.

In 16.3 above, called "application of the function M to the argument N." Usually, parentheses are used to separate the function from the argument, like so: $M(N)$. However, in λ-calculus the parentheses are used as grouping symbols. Thus, it is safe to write the function and the argument adjacent to one other. Consider the λ-expression:

$$PQR \tag{16.4}$$

where application of the function $P\ Q$ – which is itself the application of the function P to the argument Q – to R. If we do not use parentheses to separate function and argument, how are we to disambiguate expressions that involve three or more terms, such as "PQR" ? Recall our convention that we are to understand such expressions by working from left to right, always putting parentheses around adjacent terms. Thus, "PQR" should be understood as $(PQ)R$. The $PQRS$ is $((PQ)R)S$. The expression $(PQ)R$ is disambiguated; by our convention, it is identical to PQR. The expression $P(QR)$ is also explicitly disambiguated; it is distinct from PQR because it is the application of P to the argument QR (which is itself the application of the function Q to the argument R) [stnfr21].

Example 16.2 *Increment the variable x.*

The code for this is,

$$\lambda x.x+1, \tag{16.5}$$

however, the above is not an example of pure Lambda calculus, as '+' operator is not in pure lambda calculus. It defines a function of one argument whose formal parameter is x and its body is "$x+1$". To compute the function, we supply the argument 5. The entire expression, including the value of argument, is called em lambda expression.

$$(\lambda x.x+1)5 \Rightarrow 5+1=6.$$

We consider two terms identical if they only differ in names of bound variables, and denote this with $\equiv$, e.g., $\lambda x.y\ x \equiv \lambda z.y\ \ z$.

16.3.1 Reduction

The essence of *lambda*-calculus is embodied in reduction (called β-conversion) rule, which equates,

$$(\lambda x.M)N = M[x := N], \tag{16.6}$$

where $M[x := N]$ denotes the result of substituting N for all free occurrences of x in M. The M is called *body* of the lambda expression.

Definition 16.3 *A term with no possible β-reduction, i.e., no subterm of the form $(\lambda x.\ M)N$, is called* normal form.

The terms may be viewed as denoting computations of which β-reduction form the steps, and which may halt with a normal form at the end of the result.

The Church's lambda notation allows the definition of an *anonymous function*, that is, a function without a name: for example, $\lambda n.n^3$ defines the function that maps each n in the domain to n^3. We say that the expression represented by $\lambda n.n^3$ is the value bound to the identifier "cube". The number and order of the parameters to the function are specified between the λ-symbol and an expression. For instance, the expression n^2+m is ambiguous due to the definition of a function rule:

$$(3,5) \mapsto 3^2+5=14$$

or

$$(3,5) \mapsto 5^2+3=28.$$

Lambda notation resolves the ambiguity by specifying the order of the parameters:

$$\lambda n.\lambda m.n^2+m$$

or

$$\lambda m.\lambda n.n^2+m.$$

Thus,

$$\begin{aligned}(\lambda n.\lambda m.n^2+m)3\ 5 &= (\lambda n.n^2+3)\ 5\\ &= (5^2+3)\\ &= 28.\end{aligned}$$

Following are some examples of β-reduction:

- The variable x does not β -reduce to anything. It is simply a variable, and not an application term whose left-hand side is an abstraction term.
- $(\lambda x.x)a \rightarrow_\beta a$
- If x and y are distinct variables, then $(\lambda x.y)a \rightarrow_\beta y$
- The λ-term β-reduces to same value through two different steps (due to Church-Rosser property):
  1. $(\lambda x.(\lambda y.xy))ab \rightarrow_\beta (\lambda y.by)a \rightarrow_\beta ba$, and
  2. $(\lambda x.(\lambda y.xy))ab \rightarrow_\beta (\lambda x.xa)b \rightarrow_\beta ba$.

16.3.2 Expressions in the λ-Calculus

Expressions in the λ-calculus are written in strict prefix form, that is, there are no infix or postfix operators (such as $+, -$, etc.). Furthermore, function and argument are simply written next to each other, without brackets around the argument. So, where the mathematician and the computer programmer would write "$sin(x)$", in the λ-calculus we simply write "*sin x*". If a function takes more than one argument, then these are simply lined up after the function. Thus "$x+3$" becomes "$+\ x\ 3$", and "x^2" becomes "$*\ x\ x$". Brackets are employed only to enforce a special grouping. For example, where we would normally write "$sin(x)+4$", the λ-calculus formulation is "$+\ (sin\ x)\ 4$".