Algorithms, The λ Calculus and Programming

An Intuitive Approach

Abhijat Vichare

I develop the intuition behind the λ calculus and connect it to computer programming through some basic examples. This fills an often felt gap that teachers and students find between the formal structure of the λ calculus, and the principles and practice of functional programming.

Abhijat Vichare was a Senior Scientist at Computational Research Laboratories, Pune. Prior to this, he has explored GCC internals at IIT Bombay and has developed a model to understand the architecture of GCC. He has also taught at the Department of Computer Science, Pune University. He is interested in programming languages and functional programming.

Fortran, Pascal, C and C++ are some of the most commonly used languages for programming computers. Despite the differences between them, all of them actually have common roots. They all belong to the class of *imperative* languages. The framework of computer programming that these languages present is called *imperative* programming. There are other paradigms of programming. *Functional* programming is another powerful, clean and mathematically elegant paradigm. For some excellent introduction to functional programming see [1], [2] which use Haskell, and [3] which introduces and uses Scheme. For another excellent introduction to Scheme see [6].

The λ calculus forms the formal base of the central ideas of functional programming. It is a model of computation equivalent to the Turing Machine. It is useful for mathematical analysis of programming languages as well as advanced computer science. While the formal treatments of the λ calculus abound, there is a lack of intuitive presentations of the basic ideas involved. This article tries to develop the basic intuition behind the λ calculus, and connects it to functional programming. The intuitive discussion of the mathematical basis can help bridge the gaps between the practitioners and the theorists, and between the teachers and the students.

Keywords
λ calculus, functional programming, scheme.

Alternate paradigms are better accessible if we understand the intuition behind their formalism.

This article is similar in spirit to the article *Algorithms as Machines* by Kamal Lodhaya (see [4]) that has appeared in this journal before.

1. Algorithms

The idea of an algorithm had been intuitively well understood by the early twentieth century. During the early years of the twentieth century mathematicians and logicians started dealing with questions like: *What is an effective procedure*?, and *What is a function*?. These apparently distinct questions started as distinct formal models. A little later a surprise emerged that all of these were only superficially different. Deep within, each of them formally described the idea of an algorithm. A Turing machine view is one formal model, and the λ calculus is another. Historically, the λ calculus was developed by Alonzo Church and others to study functions. The syntax and the conversion rules attempted to formalize the notion of a 'function'. However, as Alan Turing showed later, this notion captured the same class of functions that were described by Turing machines. Hence the λ calculus also served as an alternate model for algorithmic behaviour. This article therefore could also be named: *Algorithms as Functions*!

We know a *function* as a rule that maps one object from a set to one another object in possibly another set. On the other hand, we informally know an algorithm, or an *effective procedure*, as: a step-by-step procedure that describes the *solution process* for a given class of problems. When applied to a given problem from that class, an effective procedure yields the answer. The steps are finite in number and the use of the algorithm is expected to terminate in a finite amount of time. Thus to add two integers written in the decimal system we follow the steps shown in *Figure* 1.

The algorithm in *Figure* 1 captures the essence of the solution process of the 'addition' class of problems. As

1. Place the integers one below the other such that their least significant digits are aligned.
2. Start from the least significant digits and proceed towards the most significant digit.
3. Add the digits at the current place along with any carry-forward digit.
4. Write the least significant digit of the sum just below the digits being added.
5. If a carry occurs then place it in the next higher significant place.
6. Move to the next significant place.
7. Continue step 3 until no more digits are left.

Figure 1. Algorithm to add two decimal numbers.

presented, it exhibits some interesting features:

• The number of steps are finite.

• There is no limit on the number of digits that the integers may have to add successfully. In fact, it is independent of the 'size', i.e. the number of digits, of the integers that it receives for addition.

• The time taken to obtain the sum is finite.

• Except for the knowledge of adding at most three digits, it is extremely 'mechanical' in operation. There is no 'smartness' required to produce the result once the algorithm is given.

These features reflect our intuitive understanding of the concept of an algorithm. Shortly, we will show the addition of three digits in complete detail so that the entire addition algorithm becomes totally 'mechanical'.

The central idea of an algorithm is that the details of the solution process are so clearly known that even a machine can do it.

Algorithms are valuable mental constructs since we can run them in our mind and obtain the result without actually manipulating the physical world. If the algorithm

There are many operations that are *not* algorithmic. For instance, $\lim_{x\to\infty}\frac{1}{x^2}$ is such an operation. It is well defined but is not algorithmic.

is correct then we are sure that *if* we did manipulate the physical world *then* it would be precisely in the state corresponding to the answer. Thus, according to the algorithm above, the sum of 95 and 14 is 109. If we took 95 pebbles and 14 pebbles and added them we would have 109 pebbles!

There are many operations that are *not* algorithmic. For instance, $\lim_{x\to\infty}\frac{1}{x^2}$ is such an operation. It is well defined but is not algorithmic. It is not clear if the number of steps required to obtain the limit are finite. Indeed, the number of steps may not even be countable. The number of terms in the sequence is infinite. As a consequence if we try a very algorithmic way to obtain the limit we have to keep on examining an infinite set of terms. This implies that an infinite time would be required to obtain the result. Unless some intelligence is employed to leap to the limit value, it is not possible to arrive at the result. Zeno's paradoxes elegantly bring out the non-algorithmic nature of such operations.

2. The λ Calculus

Let us take a second, more relaxed, look at the addition algorithm in the previous section. Consider the essential part of step 3: "`Add the digits at the current place`". We have some important words or phrases: Add, digits and current place. Each word or phrase is an answer to some important question. At step 3 we need to answer questions like: What to do?, On whom to do? and Where to do?. In general any algorithm answers such questions at each step. Sometimes the answers are obtained directly. Thus 'What to do?' is answered directly as 'Add' in step 3. At other times the answers are obtained as a result of the other steps of the algorithm. Thus 'Where to do?' (i.e., the 'current place') is answered by the other steps in the algorithm. In the above algorithm it is mainly answered by step 6.

The second relaxed look at the addition algorithm brings out some general features of algorithms.

• Algorithms need an ability to *identify* entities. Some entities like the 'Add' operation are active in the sense that they 'do' something. Other entities like 'digits' or 'current place' are passive in the sense that active entities act on them.

Identify, *transform* and *connect* are the essential capabilities required for the description of an algorithm.

• Some entities *transform* other entities into some other entity. Thus, *given* the 'digits', the 'Add' entity transforms them into other 'digits' – their sum.

• It must be possible to *connect* one entity to another entity. We need the ability to *give* the actual 'digits' entity to the 'Add' entity so that step 3 in *Figure* (1) is effective.

Thus the three keywords – identify, transform and connect – form the essential capabilities that are required for the description of an algorithm. It does not matter how we identify as long as each distinct entity is identified uniquely. As human beings we use *names* to identify. Thus, Add is a name, digits is another name. The transformation ability describes the changes in a given entity. It is typically a rule (or a set of rules) that tells us how to change a given object. Note that the entity is assumed to be given and then the 'how to' of the change is described. For a transformation to be effective, we need the ability to connect the transforming entity to the entity to be transformed.

To define the fundamental formal syntax of the λ calculus [see 5], all that we do is to symbolically express the three keywords above, namely identify, transform and connect. The syntax of the λ calculus is defined in terms of the notion of a λ term, defined in *Figure* 2.

A *variable* corresponds to the ability to identify, or name, objects. A *function abstraction* corresponds to the ability to transform. An *application* corresponds to the ability to connect.

A *variable* corresponds to the ability to identify, or name, objects. A *function abstraction* corresponds to the ability to transform. An *application* corresponds to the ability to connect. Note that in function abstraction the

Figure 2. The syntax of the λ calculus.

A λ term is defined as:

1. A *variable*, denoted by x, is a λ term.
2. A *function abstraction* of a function $f(x)$, denoted by $\lambda x.f(x)$, is a λ term. $f(x)$ is called the *body* of the abstraction.
3. An *application* of a λ term M to another λ term N, denoted by $(M \quad N)$, is a λ term.

notation $f(x)$ is simply some suitable expression to express transformation. The x in $f(x)$ is the *name* of the object that must be given later when the transformation is put to use. The prefix λx emphasises this place holding nature of x and makes it explicit. The '.' separates this explicit specification from the actual transformation body $f(x)$. Since the x acts as a place holder in the body $f(x)$, it is called the *bound variable.* Variables that are not bound in an expression are called *free variables.* Familiar examples of expressions that contain bound variables are: $\sum_i^N i^2$ where, i is the bound variable, or $\int_{x=x_1}^{x=x_2} f(x)\mathrm{d}x$, where x is the bound variable. The body of the abstraction is the range over which the bound variable is effective. This range is called the *scope* of the bound variable.

The most important rule is the β rule that specifies the λ term to be obtained when a λ term *M* is applied to another λ term *N*. The rule states that every occurrence of the bound variable of *M* in its body is replaced by the λ term *N*.

The behaviour of the above syntactic structure is captured in three rules of conversion called the α, β and η rules. These rules give 'meaning' to the operation of the syntax by specifying what λ term should result due to the syntax. For our purposes the most important rule is the β rule that specifies the λ term to be obtained when a λ term M is applied to another λ term N. The rule states that every occurrence of the bound variable of M in its body is replaced by the λ term N. As an example let $M \equiv \lambda x.(x+2)$ and $N \equiv 3$ be two λ terms. The λ

term M is an abstraction, the expression $(x + 2)$ in M is the *body* of that term, and x is the variable. The λ term M can be read as: Given some x yield the result of adding '2' to it. Applying M to N, i.e. $(M \quad N)$ is $((\lambda x.(x + 2)) \quad 3)$. The β rule is now used to 'process' this application: *Every x in the body of M is replaced by* 3. After the β reduction the application simply reads as: $(3 + 2)$, which evaluates to 5.

The λ calculus is deceptively simple.

3. Computation and λ Calculus

The λ calculus sketched above appears so simple that it seems to be far away from computation. I will now demonstrate using a number of simple examples that the λ calculus actually is an extremely powerful way of looking at computer programs. The examples will show that the main skill is in the ability to write the λ terms, particularly function abstractions, for various programming tasks. The examples are elementary but instructive.

3.1 *Boolean Values and Operations*

Boolean algebra has two values, $\top$ and $\perp$ (usually represented as TRUE/1 and FALSE/0 respectively) and operations like $\neg$ (negation), $\wedge$ (logical AND), $\vee$ (logical OR), etc. We usually tend to view values as 'passive' data objects. However, if we reflect for a moment then we can change our perspective to view these values as 'active'. Formally, the Boolean values TRUE and FALSE do not have any meaning in the conventional sense. In fact, they are just two different values (which is emphatically stressed by using the symbols $\top$ and $\perp$, respectively). These values actually help us to select between two different choices. Thus if we have two objects to choose from then we can imagine a Boolean value, say TRUE, as the entity that helps us to actively select the first of the given two. We then say that the act of applying TRUE to the two objects should yield the first of the two. This directly gives us the construction of the λ

The Boolean values TRUE and FALSE do not have any meaning in the conventional sense. These values actually help us to select between two different choices.

term for the TRUE object as:

$$\textsc{True} : \lambda x.\ (\lambda y.\ x) \tag{1}$$

We can read (1) as: *given* two objects x first and then y, TRUE is the *name* of the λ term that yields the first object x as a result of its transformation activity. Observe that $\lambda y.x$ is the body of the λ term: $\lambda x.\ (\lambda y.\ x)$. The parentheses can be used to clearly denote the body but can be omitted if the grouping is clear enough. Thus we can also write: $\lambda x.\ \lambda y.\ x$. We can read this active behaviour of TRUE as: given object x and object y, TRUE chooses, or *returns*, x. Finally: TRUE is just the *name* of the λ term appearing on the right-hand side of (1) that captures the behaviour of the TRUE value.

If TRUE is to yield the first of the two objects, then FALSE should yield the second. Hence, the λ term for FALSE is constructed as:

$$\textsc{False} : \lambda x.\ \lambda y.\ y \tag{2}$$

We can read this active behaviour of FALSE as: given object x and object y, FALSE must choose y (since TRUE has been constructed to choose x).

We use β rule to apply TRUE to two given objects, say apple and orange, as follows:

$$\begin{aligned}&((\textsc{True}\ \texttt{apple})\ \ \texttt{orange})\\ &\qquad\equiv (((\lambda x.\ (\lambda y.\ x))\ \ \texttt{apple})\ \ \texttt{orange})\\ &\qquad\equiv ((\lambda y.\ \texttt{apple})\ \ \texttt{orange})\\ &\qquad\equiv \texttt{apple}\end{aligned} \tag{3}$$

Similarly FALSE yields:

$$\begin{aligned}&((\textsc{False}\ \ \texttt{apple})\ \ \texttt{orange})\\ &\qquad\equiv (((\lambda x.\ (\lambda y.\ y))\ \ \texttt{apple})\ \ \texttt{orange})\\ &\qquad\equiv ((\lambda y.\ y)\ \ \texttt{orange})\\ &\qquad\equiv \texttt{orange}\end{aligned} \tag{4}$$

Is the object 'apple' a λ term ? Yes, it is. It is just a name of an object – a variable. The object 'orange' is also a λ term. In fact any other type of λ term can occur in place of any or both of these (illustrative) λ terms.

A conditional expression λ term yields the value of one of two λ terms depending on the Boolean value of a condition λ term.

We can now construct a more useful λ term: the *conditional expression*. An expression yields a value. A conditional expression λ term yields the value of one of two λ terms depending on the Boolean value of a condition λ term. A conditional expression thus needs to be given three λ terms: the condition term, say b, and the two other λ terms, say m and n, between which a selection is to be made. Observing that b will either be the λ term TRUE or the term FALSE, we can construct the conditional expression by simply applying b to m and n. Naming the conditional expression as IF, we write it as:

$$\textsc{If} : \lambda b.\ \lambda m.\ \lambda n.((b\ \ m)\ \ n) \qquad (5)$$

The body of the λ term in (5) is inspired from (3, 4). Languages like C have a conditional *statement*, the `if-then-else` statement. This statement only changes the control flow of the computation and does not necessarily yield a value. In contrast, the conditional expression in (5) yields a value. It is in no way concerned about control flow. In fact, it is concerned only with the question of which one (and only one) of the two expressions to evaluate next. It corresponds to the `_?_:_` ternary operator in C.

Given the construction of the conditional expression, the Boolean operators are now easily constructed. The logical negation operator, NOT, is easily constructed as:

$$\textsc{Not} : \lambda x.\ (\textsc{If}\ \ x\ \ \textsc{False}\ \ \textsc{True}) \qquad (6)$$

where IF, TRUE and FALSE are λ terms as constructed earlier. The thought process behind the construction of (6) is quite simple: if x has the value TRUE, then the conditional expression in (6) should yield FALSE and vice versa.

It is instructive to construct the λ terms for the other Boolean operations like AND, etc.

3.2 *Natural Numbers*

We know that the natural numbers 0, 1, 2, ... are also called as 'counting' numbers. To view them from the λ calculus perspective, we again need to change over from a rather passive view of numbers as 'data' to an active view of numbers. This is possible by focusing on the counting activity that numbers are used for. Imagine counting a few chairs by pointing a finger sequentially at each one. A 'finger' is *applied* to every object that satisfies the criterion of being a 'chair'. A counting number is simply the number of times the *application* of one object, the finger in this concrete case, to another object, the chair, is successfully carried out. Now observe that the object being applied need not be something like the finger. It can be another active process too. For instance, imagine the process that doubles the object it is given. We can *apply* the '`double-up`' activity to a chair and get two chairs. If we apply the '`double-up`' activity to each of N chairs, then we get $2N$ chairs in all. However, the activity of doubling has been *applied* only N times. Thus the counting numbers are simply the number of times one object, say f, is applied to another, say x. If such an application cannot be done, then the number is '0'. We can use this insight to construct the λ expressions for natural numbers as follows (we 'name' the numbers as usual):

The counting numbers are simply the number of times one object, say *f*, is applied to another, say *x*. If such an application cannot be done, then the number is '0'.

$$\begin{aligned}
0 &: \lambda f.\ \lambda x.\ x \\
1 &: \lambda f.\ \lambda x.\ (f\ \ x) \\
2 &: \lambda f.\ \lambda x.\ (f\ \ (f\ \ x)) \\
3 &: \lambda f.\ \lambda x.\ (f\ \ (f\ \ (f\ \ x))) \\
\dots & \\
n &: \lambda f.\ \lambda x.\ (f\ \dots (f\ \ x) \dots) \qquad (7)
\end{aligned}$$

The above way of looking at numbers was proposed by

Alonzo Church and hence these are also called as *Church numerals*.

We observe that a number in the λ calculus sense is simply the number of applications of some entity, f, to another entity, x. While we have used the act of 'point a finger' as the entity, the f can be any reasonable process. For instance, f can be the process of doubling. Since bound variables can be renamed systematically – and that is what the α conversion rule is about – we can write: $n \equiv \lambda f.\ \lambda x.\ (f\ \ldots (f\ \ x) \ldots) \equiv \lambda g.\ \lambda y.\ (g\ \ldots (g\ y) \ldots)$. The idea that a natural number n is simply n applications of some (λ term) f to some (λ term) x can be expressed as the following application λ term:

We observe that a number in the λ calculus sense is simply the number of applications of some entity, *f*, to another entity, *x*.

$$\begin{aligned}
&(\lambda m.(\lambda g.\lambda y.\ (m\ \ g\ \ y))\quad n) \\
&\quad \equiv \lambda g.\ \lambda y.\ (n\ \ g\ \ y) \qquad \text{[application, } \beta \text{ rule]} \\
&\quad \equiv \lambda g.\ \lambda y.\ ((\lambda f.\ \lambda x.\ (\underbrace{f\ \ldots (f}_{\text{n times}}\ \ x) \ldots))\ \ g\ \ y) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{[definition of } n\text{]} \\
&\quad \equiv \lambda g.\ \lambda y.\ ((\lambda x.\ (\underbrace{g\ \ldots (g}_{\text{n times}}\ \ x) \ldots))\ \ y) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{[application, } \beta \text{ rule]} \\
&\quad \equiv \lambda g.\ \lambda y.\ (\underbrace{g\ \ldots (g}_{\text{n times}}\ \ y) \ldots) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{[application, } \beta \text{ rule]} \\
&\quad \equiv \lambda f.\ \lambda x.\ (\underbrace{f\ \ldots (f}_{\text{n times}}\ \ x) \ldots) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{[renaming, } \alpha \text{ rule]} \\
&\quad \equiv n \qquad\qquad\qquad\qquad\qquad \text{[definition of n]}
\end{aligned} \tag{8}$$

This observation can motivate the definition of the successor function called *succ*. It takes a natural number n and returns the next natural number $(n+1)$. The λ

abstraction for the *succ* function can be written as:

$$\text{succ} :: \lambda n.\ \lambda g.\ \lambda y.\ ((n\ g)\ (g\ y)) \qquad (9)$$

Note that g and y are bound variables and hence can be renamed systematically. To see that the *succ* definition indeed behaves as expected, let us calculate '(succ 1)'.

$$\begin{aligned}
(\text{succ } 1) &\equiv (\lambda n.\ \lambda g.\ \lambda y.\ ((n\ g)\ (g\ y))) \quad 1) \\
&\equiv (\lambda g.\ \lambda y.\ ((1\ g)\ (g\ y))) \\
&\equiv (\lambda g.\ \lambda y.\ (((\lambda f.\ \lambda x.\ (f\ x))\ \ g)\ (g\ y))) \\
&\equiv (\lambda g.\ \lambda y.\ ((\lambda x.\ (g\ x))\ (g\ y))) \\
&\equiv (\lambda g.\ \lambda y.\ (g\ (g\ y))) \\
&\equiv (\lambda f.\ \lambda x.\ (f\ (f\ x))) \\
&\qquad [\alpha \text{ conversion. Rename : } g \text{ by } f \text{ and } y \text{ by } x] \\
&\equiv 2 \qquad (10)
\end{aligned}$$

The intuition behind Church numerals can be extended to describe the addition operation. Addition of Church numerals m and n is simply the Church number obtained when n applications are *continued* after the first m applications. Using the ideas in the definition of the *succ* function, we write the definition of addition as:

$$\text{add} :: \lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ ((m\ f)\ (n\ f\ x)) \qquad (11)$$

The 'add' λ term in (11) captures the essential behaviour of the addition operation, '+', over natural numbers.

Addition of Church numerals *m* and *n* is simply the Church number obtained when *n* applications are *continued* after the first *m* applications.

The other operations like subtraction and recursion can be defined but need more conceptual framework to proceed. We will skip them but mention that the two interesting concepts needed are the predecessor function that yields the previous natural number of the given natural number, and a peculiar λ expression Y that has the property: $(Y\ E) = (E\ (Y\ E))$. The Y is called the fixed point operator and is needed to capture recursion.

3.3 *Pairs*

An ordered *pair* of two objects o_1 and o_2 is denoted as $(o_1 \;.\; o_2)$ and is just some 'gluing' of the objects such that the first of the pair is the object o_1 and the second is the object o_2. We wish to glue two objects together so that we can process them together. The pair (*name, age*) that captures the name of a person *and* his/her age together suggests the need to treat the composite object as a single entity. In programming, pairs serve to create structures of objects. The objects can be any λ terms. How do we construct the λ term for a pair?

We wish to glue two objects together so that we can process them together. The pair (*name*, *age*) that captures the name of a person *and* his/her age together suggests the need to treat the composite object as a single entity. In programming, pairs serve to create structures of objects.

Given that the pairing process simply prepares the two objects for eventual processing by some other object, we can propose the following definition for a pair $(m \;.\; n)$ of two objects m and n:

$$\text{pair} : (m \;.\; n) : \lambda m.\ \lambda n.\ \lambda f.\ ((f\ m)\ n) \qquad (12)$$

If we wish to extract the first of the pair, then we must arrange the λ expression so that we can *select* the first object. We already know how to select the first of two given objects. We must have the TRUE λ expression from (1) take the place of the f in (12). Since this must occur *after* β conversion, the selection activity must be presented as the 'eventual' process that acts on the given pair. Thus the definition of *first* operator that yields the first of the pair p can be written as:

$$\text{first} : \ : \lambda p.(p\ \textsc{True}). \qquad (13)$$

Naturally, the definition of the *second* operator that yields the second of the pair p can be written as:

$$\text{second} : \ : \lambda p.(p\ \textsc{False}). \qquad (14)$$

If we wish to pair up two objects, 1 and 2, to obtain the pair (1 . 2) then

$$\begin{aligned}(1\ .\ 2) &\equiv ((\text{pair } 1)\ 2)\\ &\equiv (((\lambda m.\ \lambda n.\ \lambda f.\ ((f\ m)\ n))\ \ 1)\ \ 2)\\ &\equiv ((\lambda n.\ \lambda f.\ ((f\ 1)\ n))\ \ 2)\\ &\equiv (\lambda f.\ ((f\ 1)\ 2)) \qquad (15)\end{aligned}$$

The RHS of the last line of (15) is the final λ term that denotes the pair (1 . 2). Note that it expects that some f is to be given later. We can now see how the definition of *first* allows us to extract the first of the pair (1 . 2).

$$\begin{aligned}(\text{first }(1\ .\ 2)) &\equiv ((\lambda p.(p\ \textsc{True}))\ (1\ .\ 2))\\ &\equiv ((1\ .\ 2)\ \textsc{True})\\ &\equiv ((\lambda f.\ ((f\ 1)\ 2))\ \textsc{True})\\ &\equiv ((\textsc{True}\ 1)\ 2)\\ &\equiv (((\lambda x.\ \lambda y.\ x)\ 1)\ 2)\\ &\equiv ((\lambda y.\ 1)\ 2)\\ &\equiv 1 \qquad (16)\end{aligned}$$

3.4 *Some Observations*

We have seen the λ calculus in action in the above sections. We can now distill some interesting concepts that emerge although our view has been quite limited.

• Entities that we often imagine as passive are also active entities. The intuition behind objects like TRUE, FALSE, church numerals, etc., show that objects we often take as passive do actually have an active view too. One mental paradigm shift is in appreciating that all entities assume active or passive roles depending on where we choose to place them.

One mental paradigm shift is in appreciating that all entities assume active or passive roles depending on where we choose to place them.

• The *syntax* of the λ calculus specifies what are λ terms and how they are arranged. The parentheses in the syntax for function application are important. The first term of an application is the 'operator', the entity that acts, and the second term is the 'operand', the entity that is acted upon. Thus the syntax is a simple prefix notation: operator followed by operands – always.

• The conversion rules, of which we have seen the β rule mainly and a little of the α rule, specify the operational effects of the syntax.

• The results of applying the conversion rules are other λ terms which we can continue converting. Does this process of conversions ever stop? Although we have not stated explicitly, the Church–Rosser theorem assures us that if the conversions indeed can stop, then we can keep on applying them *in any order* until no more conversions are possible. Such a λ term which cannot be converted further is said to be in *normal form.*

Church–Rosser theorem assures us that if the conversions indeed can stop, then we can keep on applying them *in any order* until no more conversions are possible. Such a λ term which cannot be converted further is said to be in *normal form*.

• There is no real need to distinguish between passive data and active processes as we have seen while constructing the λ terms for various data values! This property is called *first classness* and is in stark contrast to the division between data and code in our usual languages. First classness allows us to treat data and processes on the same footing. Thus just like data, processes can be passed as arguments to other processes, or can be returned from other processes, or can be a part of some other structures. In fact, there is no need to have separate concepts of 'data structures' and 'code structures'. Come to think of it such a division in usual languages is actually artificial. The memory of a computer only has integers in binary representation and there is no mechanism to distinguish if a certain value in memory is a data or a code (i.e., a CPU instruction) unless the values of the instruction pointer and data pointer in the CPU are given.

First classness allows us to treat data and processes on the same footing. Thus just like data, processes can be passed as arguments to other processes, or can be returned from other processes, or can be a part of some other structures.

• The β rule is just text substitution. The λ calculus effects computation through substitutions. An interesting effect of text substitution *and* application is that one can construct 'partial evaluations'. Thus a function, i.e.

The β rule is just text substitution. The λ calculus effects computation through substitutions. An interesting effect of text substitution *and* application is that one can construct 'partial evaluations'.

an abstraction, that expects two arguments can be given only one argument. The β rule then substitutes the argument for the corresponding parameter. This yields a λ term with only one remaining parameter. The function is said to have been partially evaluated and results in a λ term that is ready for further application to another λ term that will be the argument not supplied before. This is a powerful ability and has no analogue in the imperative languages. We will see some glimpses of the power of this ability.

For example, normally the addition operation requires two numbers as its arguments. However, the `add` λ term in (11) can do useful partial evaluation when given only one argument! Suppose we give only one argument with value '1' to the add λ term. Then, observing that 1 is the *name* of the λ term $\lambda g.\ \lambda h.\ (g\ h)$ – the Church numeral 1 with f renamed as g and x renamed as h, we have:

$$\begin{aligned}
(\text{add}\ \ 1) &\equiv ((\lambda m.\ \lambda n.\ \lambda f.\ \lambda x.\ ((m\ f)\ (n\ f\ x))) \quad 1) \\
&\quad\ \text{[Replace every occurrence of } m \text{ by 1]} \\
&\equiv (\lambda n.\ \lambda f.\ \lambda x.\ ((1\ f)\ (n\ f\ x))) \\
&\quad\ \text{[Replace 1 by its } \lambda \text{ term]} \\
&\equiv (\lambda n.\ \lambda f.\ \lambda x.\ ((\lambda g.\ \lambda h.\ (g\ h)\ f)\ (n\ f\ x))) \\
&\equiv (\lambda n.\ \lambda f.\ \lambda x.\ ((\lambda h.\ (f\ h))\ (n\ f\ x))) \\
&\equiv (\lambda n.\ \lambda f.\ \lambda x.\ \quad \underbrace{(f\ (n\ f\ x))}_{\text{one more application of } f} \quad) \qquad (17)
\end{aligned}$$

Not only are partial evaluations possible, but they result in a remarkable ability to create processes 'on the fly' in practical programming.

The last term in (17) has finished applying the λ term for the Church number 1 and is 'waiting' for the *next* argument n of the addition operation. We now have an operator which would add 1 to any number given to it. Most modern CPUs have an instruction, `inc`(rement), that adds 1 to its argument. Equation (17) is the λ term for this instruction, and we now understand it as a simple effect of partial evaluation of addition when given only one argument. This is called *currying*. Not only are partial evaluations possible, but they result in a remarkable ability to create processes 'on the fly' in practical programming.

Finally, we already know the operation in (17) as `succ` – see (9). But these two expressions look very different. They are not. Try `(succ 1)` and `((add 1) 1)`.

The λ calculus tells us that we need the ability to *bind* a name to a value, i.e., just associate the name to a λ term. Once bound, the value does not change.

• The λ calculus tells us that we need the ability to *bind* a name to a value, i.e., just associate the name to a λ term. Once bound, the value does not change. This is remarkable. Our usual programming languages do such things differently. They use an operation called *assignment*. An *assignment* operation identifies a location – an *address of a store* – into which values may be stored, and associates a name with the location and not the value. A name is no longer bound to a value. The value can be changed without affecting the association between the name and the address. We say that the value can be *mutated* to another value by overwriting the previous value at the location associated with the name. Practical programming using usual languages based on the Turing machine use the assignment operation intensively. However, the λ calculus shows that there is no need of the assignment operation for computation purposes. Instead, just binding is enough when supported by the abilities to yield λ terms through abstraction and application.

• Programming languages can almost directly be created by devising a computer representable syntax of the λ calculus. For instance we might use the syntax '`lambda`' on a computer for λ.

Languages like Haskell are able to actually deduce the types of objects that occur in an expression, and the programmer is not required to specify the types.

• Technically, the λ calculus sketched above is an inconsistent theory. Extensions like the typed λ calculus have been developed and have advanced the art to very fine levels. For instance, languages like Haskell are able to actually deduce the types of objects that occur in an expression, and the programmer is not required to specify

Scheme is almost a direct computer representation of the λ calculus.

the types. In practice, one uses such extended versions. For instance, the untyped calculus above is augmented with predefined data types (like Booleans or numbers) and operations.

4. Programming and the λ Calculus

The language Lisp, particularly its dialect Scheme, is almost a direct computer representation of the λ calculus. In Scheme, a 'name' is bound to a 'value' using the '`define`' construct. To represent λ on a computer the word '`lambda`' is used. Scheme calls processes as *procedures*. Since data and procedures are first class in Scheme, the '`define`' syntax can be used to associate names with procedures or values. Thus:

• The Scheme expression `(define pi 3.14159)` binds, i.e. associates, the name `pi` with the value `3.14159`. This is an example of the first syntax rule of the λ calculus: the ability to name objects.

• The Scheme expression `(lambda (x) (* x x))` corresponds to the λ calculus syntax of *function abstraction*: $\lambda x.\ (*\ x\ x)$. It is a process that when given an object multiplies it by itself. Note that function abstraction merely captures the *processing* to be done, and does **not** give a name to the process.

• We can use the ability to name objects and write the Scheme expression: `(define square (lambda (x) (* x x)))` to associate the name '`square`' with the process of taking an object and multiplying it by itself. It is interesting to see the uniformity of syntax that emerges from first classness: a value 3.14159 or a process $\lambda x.\ (*\ x\ x)$ can be associated with their names using just one construct – the `define`. However, Scheme also provides an alternate syntax for defining procedures in a more conventional way. The above procedure can also be written in Scheme as: `(define (square x) (* x x))`. Finally, note that the body of the abstraction, `(*`

`x x)`, can be any other λ term, however complex.

• To use the squaring procedure, we need to *apply* it to some object. The Scheme expression: `(square pi)` applies the λ term (bound to the name) `square` to the λ term (bound to the name) `pi`. To evaluate this expression, one conceptually performs β conversion. Thus `square` and `pi` are substituted by their bindings. This gives: `((lambda (x) (* x x)) 3.14159)`. This can be further reduced to: `(* 3.14159 3.14159)`. Applying the '`*`' operation to its arguments yields: `9.869588` which is the answer since it cannot be converted further.

The basic correspondence between the formal sketch of the λ calculus and a practical programming language, Scheme is now in place.

The basic correspondence between the formal sketch of the λ calculus and a practical programming language, Scheme is now in place. We will now finish this section with a set of Scheme programs that illustrate some of the observations in Section 3.4 as well as some other exciting and practical concepts.

4.1 *Illustrative Scheme Programs*

Basic Recursion: Consider the classic `factorial` function defined recursively as:

$$\begin{aligned} \forall n \in \mathbb{N}, \quad n! &= 1, \quad n = 0 \\ &= n \times \ (n-1)!, \text{ otherwise}, \quad (18) \end{aligned}$$

we can write a Scheme procedure as:

```
(define fact
   (lambda (n)
      (if (= n 0)
          1
          (* n (fact (- n 1))))))
```

Try `(fact 5)` to obtain the answer 120. Note that the '`if`' is the conditional *expression* as noted in (5). Also, expressions like `(= n 0)` are in prefix form. In their more familiar infix form we write them as:

It is not easy to write a C function that takes another function as an argument and uses it for evaluation.

`n = 0`. The example also illustrates some practical programming styles of indentation and using new lines.

Procedures as Arguments: Compare the expressions: $\sum_{i=1}^{10} i^2$ and $\sum_{i=1}^{10} i^3$. They sum the squares and cubes of integers between 1 and 10. In general the summation notation, $\sum_{\dots}^{\dots}$ performs addition of values of any function, like i^2 or i^3, given to it. There is a clear separation between the repeated operation, addition, and the values to operate on as generated by some function, say i^2 or i^3. In languages like C it is not easy to capture this separation. That is, it is not easy to write a C function that takes another function as an argument and uses it for evaluation. Here is a Scheme procedure to perform summation on arbitrary functions that it receives as argument.

```
(define summation
   (lambda (start end function)
      (if (> start end)
          0
          (+ (function start)
             (summation (+ start 1)
  end function)))))
```

Try `(summation 1 10 square)` to find the sum of the squares of the first ten numbers. Also try `(summation 1 10 (lambda (x) (* x x x)))` to sum the cubes of the first ten numbers.

The 'Pair' and 'List': In Scheme the pair, first and 'second' λ terms defined by (12), (13) and (14) are respectively called as: `cons`, `car` and `cdr` (pronounced 'coulder'). Additionally, a special object called `nil` and written as `'()` is introduced to define lists. A list is a particular kind of pairing operation that is recursively defined as:

$$list \ := \ nil \mid (\text{pair} \ \ object \ \ list) \qquad (19)$$

where we have used the BNF notation for compactness. Reading ':=' as: 'is defined as', we can read (19) as 'A list is defined as either nil or the pair of an object and another list.' Thus the pair given by: `(cons square '())` yields `(square . '())`. This is normally written as just: `(square)` and denotes a list with one element `square`. Recall that `square` is the name of the procedure that squares. We can bind this list to a name: `(define list-of-procedures (cons square '()))`. Also: `(car list-of-procedures)` yields `square`, and `(cdr list-of-procedures)` yields `'()`. Using the recursive definition of lists in (19), we can add another procedure `cube` as: `(cons cube list-of-procedures)`. This list in pair notation is: `(cube . (square .'()))`, and is normally written as `(cube square)`. The operation of creating lists is often used and hence there is a short hand in Scheme: `(list cube square)` → `(cube square)`.

A list is defined as either nil or the pair of an object and another list.

Now here is another example of first classness at work. Observe that the list structure of the expression `(cube square)` is the same list structure of the expression `(define pi 3.14159)`. For instance: `(car '(define pi 3.14159))` yields `define`. I used '`'`' (single quote) character to tell Scheme to **not** evaluate the list as we want to treat it as data. Thus Scheme expressions that we see as 'code' are themselves lists! Hence we can write Scheme procedures that *construct* code by composing lists in desired ways. In other words, we can write Scheme procedures that can manipulate other procedures by treating them as 'data', i.e., just a list of some objects! We now understand why this way of writing programs is called 'LISt Processing' and the language is called LISP. Scheme is a dialect of LISP.

We can write Scheme procedures that can manipulate other procedures by treating them as 'data', i.e., just a list of some objects!

5. Closing Remarks

The languages based on the λ calculus promote a view of programming where λ terms called operators are applied on other λ terms called operands to yield λ terms called return values. This exactly corresponds to our usual idea of a mathematical function. Mathematical functions simply take values as arguments and return values as 'answers'. This style of programming that emerges is therefore called *functional programming*. It is a distinct style of programming and requires a mental paradigm shift compared to our usual programming in C/Fortran etc. I have tried to convey this paradigm shift through a discussion of the intuitive connections between the formal system of the λ calculus and a programming language from the LISP family, Scheme.

Mathematical functions simply take values as arguments and return values as 'answers'. This style of programming that emerges is therefore called *functional programming*.

Functional programming is far more extensive than described here. My aim has been to aid the mental shift required for functional programming (FP). We have barely seen the tip of the iceberg in this article and I hope that even this glimpse has shown something more exciting than imperative languages. Far more exciting concepts and ideas proliferate the FP world. Some of the ideas that are considered 'novel' and 'innovative' in the present maturity of imperative languages, message passing in C++ for instance, have already been investigated in the FP world. Old FP concepts like tail recursion are beginning their entry in this world of imperative languages. FP tends to look at programs at a much more logical level than most popular languages. As a result most FP programs are very compact as compared to equivalent programs in procedural languages.

Functional programming tends to look at programs at a much more logical level than most popular languages. As a result most FP programs are very compact as compared to equivalent programs in procedural languages.

In this article I have addressed one reason why functional programming is difficult to understand and teach. I found that exposing the intuition behind the λ calculus makes it much more accessible to students. The λ calculus and the functional programming paradigm based on it may not be used as much in industry as languages like C or C++. The probability that one may use a FP language in his or her professional life is indeed very low. However, the perspectives that FP opens up has a deep impact on the way programs in procedural languages are designed and developed. Personally, this benefit of FP alone suffices to teach and learn these concepts.

Suggested Reading

[1] **Madhavan Mukund, A Taste of Functional Programming – I, *Resonance*, Vol.12, No.8, p.27, August 2007.**

[2] **Madhavan Mukund, A Taste of Functional Programming – II, *Resonance*, Vol.12, No.9, p.40, September 2007.**

[3] **Harold Abelson, Gerald Sussman with Julie Sussman, *Structure and Interpretation of Computer Programs*. http://mitpress.mit.edu/sicp.**

[4] **Kamal Lodhaya, Algorithms as Machines, *Resonance*, Vol.14, No.4, p.367, April 2009.**

[5] **Wikipedia, http://en.wikipedia.org/wiki/Lambda_calculus**

[6] **How to Design Programs, http://www.htdp.org**

Address for Correspondence
Abhijat Vichare
C3/701
Kumar Parisar Society
Near Gandhi Bhavan
Kothrud
Pune 411038, India.
Email: abhijatv@gmail.com