LAMBDA CALCULUS FROM LOGICAL PERSPECTIVE

LAWRENCE VU

ABSTRACT. This article presents a glimpse about the process of deriving λ-calculus from scratch which I have undertook. In most sections, the author assumes the reader basic knowledge of *set theory*, *recursive definition*, *(first-order) logic*, *functional and object-oriented programming*.

1. INTRODUCTION

I believe that most programmers with limited theoretical exposure[1] remember *"the ability to pass functions as arguments to other functions"* (or its paraphase, *"functions are treated as first class objects"*) as the only feature of functional programming. This is probably because most functional programming languages *hide many things* from programmers and shows users the superficial beauty beneath the paradigm. There are many ideas diverging from conventional programming such as

- *Everything is a function.*
- *Function has exactly one argument and has any number of arguments.*

I used to program in functional programming languages. Recently, I realize that the *anonymous function* construct (i.e. `lambda` in Scheme or `fun` in Ocaml) resembles the quantified formulas in first-order logic and thus, suspect that this construct is *sufficient* for every possible computation. Indeed, this inspiration allows me to develop the well-known (untyped) λ-calculus[2] from scratch.

As said, λ-calculus can be seen as a *computational analogue* of *first-order logic*. Therefore, reader with background in mathematical logic will find the correspondence table 1 helpful. This article partly recounts the mentioned experience. I choose to leave the corresponding *semantic* for a different occasion.

Date: 3rd March 2013.

[1]Such a group of programmer includes myself.

[2]This calculus is originally developed by Alonzo Church.

	First-order logic	**λ-calculus**
Syntax	Logical terms and formulas	λ-expressions
Manipulation system	Proof system	Rewriting system

TABLE 1. Logic to λ-calculus correspondence

The next section will cover the correspondence as described in the table 1 and the one after that illustrates how imperative programming is done in λ-calculus. Most exercises included are for inspired reader and some of them should be treated not as fact because they are derived using the author's philosophy and heuristic arguments.

2. First-order Logic to λ-expressions

2.1. **Syntax.** The basic building blocks in λ-calculus are λ-expressions. Let $V = \{v_0, v_1, v_2, \dots\}$ be a fixed collection of *variable symbols* and assume that V does not contains parentheses, comma as well as the symbol λ (view them as *reserved keywords*).

Definition 1. *The collection Λ is the* minimal *collection of strings such that*

(i) $V \subset \Lambda$*;*
(ii) *If $\phi, \psi \in \Lambda$ then $\phi(\psi) \in \Lambda$;*
(iii) *If $\phi \in \Lambda$ then $\lambda(v, \phi) \in \Lambda$ for any $v \in V$.*

Elements of Λ are called λ-expressions *(or just* expressions *if there is no confusion).*

Definition 1 seems cryptic but its meaning is very simple. Intuitively, it allows one to recursively collect more strings to Λ starting from V. Clause (i) gives the basic expressions: *every variable is an λ-expression.* Then , $v_i(v_j)$ are also λ-expressions by applying clause (ii) with all combinations of $\phi = v_i$ and $\psi = v_j$. With clause (iii), we include to Λ all strings $\lambda(v_i, v_j)$ by letting $v = v_i$ and $\phi = v_j$. Applying this deduction again, we infer that all combinations

$$v_i(v_j)(v_k), v_i(v_j)(v_k(v_l)), v_i(v_j)(\lambda(v_k, v_l)), \dots \in \Lambda$$

The process continues *ad infinitum*[3].

The *minimality* condition in the definition means that only strings obtained *via the above process* are in Λ; otherwise, Λ is not uniquely

[3]This process can be made precise by constructing the approximating sequence Λ_n with

$$\begin{aligned}\Lambda_0 &= V \\ \Lambda_{n+1} &= \Lambda_n \cup \{\phi(\psi) : \phi, \psi \in \Lambda_n\} \cup \{\lambda(v, \phi) : v \in V \wedge \phi \in \Lambda_n\}\end{aligned}$$

defined: the collection of all possible string also works. In particular, (due to minimality), if ϕ is a λ-expression then either

(i) ϕ is a variable; or
(ii) $\phi = \alpha(\beta)$ for some λ-expressions α and β; or
(iii) $\phi = \lambda(v, \alpha)$ for some variable $v \in V$ and λ-expression α.

Expressions in the third form are called a λ-*abstraction*. Before going on, I would like to draw comparison with constructions of formulas in first-order logic:

- The set of variable symbols V corresponds to the dummy logical variables.
- The second case in definition 1 corresponds to construction of new formulas using logical connective (and, or, not, etc). In particular, *if ϕ and ψ are a logical formulas then $\phi \wedge \psi$, $\phi \vee \psi$, ... are.* That said, I could have choose an alternative notation, say $\phi \cdot \psi$, instead of the familiar notation $\phi(\psi)$ for function application.
- The last part of the definition corresponds to the quantified formula. Think of, say, *if ϕ is a logical formula then $\exists v \cdot \phi$ and $\forall v \cdot \phi$ are also logical formulas.* The alternative notation $\lambda v : \phi$ would probably illustrate this analogy better.

A major difference between the two scenarios is that logical formulas are *well-typed*.

For simplicity, from now on, I will consistently use lower case letters $f, x, y, z, \ldots$ for variables instead of $v_0, v_1, \ldots$, (different letters are for different variables i.e. different element of V) and Greek letters $\alpha, \beta, \ldots$ denotes λ-expressions.

2.2. The intuition behind λ-expressions. By themselves, λ-expressions are meaningless strings of symbols but the definition certainly contain some inherent meaning. Intuitively, the expression $\lambda(x, \alpha)$ represents *"a function that map x to α"*[4] and $\phi(\psi)$ should be read *"apply the function ϕ on ψ"*.

---

for all $n \geq 0$. The idea is that Λ_n collects the expressions constructed within n steps. Then we simply collect all Λ_n to get

$$\Lambda = \bigcup_{n=0}^{\infty} \Lambda_n.$$

[4] I do not know why Church used the Greek letter λ but if I were him, I would use either τ for *"transform"* or μ for *"map"*.

A quick readers might ask: if $\lambda(x, \alpha)$ is supposed to be a function, then what is its *domain* and *co-domain*? In untyped λ-calculus, the intended domain/co-domain is Λ, the collections of λ-expressions. Mathematically speaking:

$$\begin{aligned} \lambda(x, \alpha) : \Lambda &\to \Lambda \\ \phi &\mapsto \alpha[x/\phi] \end{aligned}$$

So the story goes naturally: α gives the body (i.e. implementation) of the function $\lambda(x, \alpha)$. So to compute $\lambda(x, \alpha)(\phi)$ on its input, we simply *substitute* ϕ to the occurrences of x in α and *simplify* the resulting expression[5]. On the other hand, λ can basically be viewed as a *function constructor*: it takes in a variable symbol x, the λ-expression α and return a *function object* which (on input ϕ) returns $\alpha[x/\phi]$, the λ-expressions obtained by replacing all *free occurrences*[6] of x by ϕ.

To summarize the whole discussion, functions constructed via λ-abstractions syntactically transforms λ-expressions to further λ-expressions. Again, the central idea of λ-calculus about algorithmic sufficiency of λ-calculus (as in the introduction) is now tantamount to: the ability to define abstraction and application is computationally sufficient.

2.3. **Rewriting system.** Now, I will present *"a rewriting system"* for λ-calculus (i.e. the previously missing pieces about substitution and simplification). The role of this rewriting system, namely giving computational capability to λ-calculus, is in par with the reasoning/deduction role of *a proof system* to logic. First, definition 2 defines formally the notation $\phi[v/\psi]$ of variable substitution:

Definition 2. *Suppose that ϕ and ψ are λ-expressions. The substitution of x by ψ in ϕ, denoted by $\phi[x/\psi]$, is defined to be:*

---

[5]Substitution and simplification mentioned here will be defined shortly.

[6]The variable symbol x as in $\lambda(x, \alpha)$ should be viewed as *"dummy" placeholder*. For instance, two expressions $\lambda(y, y)$ and $\lambda(x, x)$ mean basically *the same function*, namely the identity (echo) function, which returns whatever is input. Such dummy variables are called *bound variables* of the expression. The analogous scenario in logic is about bound variable of a formula. For example, $\forall x : f(x) = x$ means the same thing as $\forall y : f(y) = y$. We formally define *the set of free variables of a λ-expression ϕ* by the following:

$$FV(\phi) := \begin{cases} \{\phi\} & \text{if } \phi \in V \\ FV(\alpha) \cup FV(\beta) & \text{if } \phi = \alpha(\beta) \\ FV(\alpha) \backslash \{v\} & \text{if } \phi = \lambda(v, \alpha). \end{cases}$$

(i) *If* $\phi \in V$ *then*

$$\phi[v/\psi] := \begin{cases} \psi & \textit{if } \phi = x; \\ \phi & \textit{otherwise.} \end{cases}$$

(ii) *If* $\phi = \alpha(\beta)$ *then*

$$\phi[x/\psi] := \alpha[x/\psi](\beta[x/\psi])$$

(iii) *If* $\phi = \lambda(v, \alpha)$ *then*

$$\phi[x/\psi] := \begin{cases} \phi & \textit{if } x = v; \\ \lambda(v, \alpha[x/\psi]) & \textit{otherwise.} \end{cases}$$

Definition 3. *Suppose that* ϕ *and* ψ *are* λ*-expressions. We say that* ϕ simplifies/reduces/is equivalent to ψ *or* ψ is derivable from ϕ*, denoted by* $\phi \vdash \psi$*, if one of the following holds:*

(i) $\phi = \psi$
(ii) $\phi = \alpha(\beta)$ *and* $\psi = \gamma(\beta)$ *and* $\alpha \vdash \gamma$
(iii) $\phi = \alpha(\beta)$ *and* $\psi = \alpha(\gamma)$ *and* $\beta \vdash \gamma$
(iv) $\phi = \lambda(x, \alpha)$ *and* $\psi = \lambda(x, \beta)$ *and* $\alpha \vdash \beta$
(v) $\phi = \lambda(x, \alpha)$ *and* $\lambda(y, \alpha[x/y]) \vdash \psi$ *and* y *is substitutable for* x *in* α
(vi) $\phi = \lambda(x, \alpha)(\beta)$ *and* $\alpha[x/\beta] \vdash \psi$ *and* β *is substitutable for* x *in* α

Clause (i) is trivial: one can leave an expression alone (the *"identity rule"*). Clause (ii)–(iv) says that one can replace *sub-expressions* by directly derivable ones[7], and I will name them *"structural rules"*. Clause (v) says that variables are just placeholders and can be replaced by any appropriate ones. I will adapt the nicer name from first-order logic, *"alphabetic invariant rule"*. The last case (which I shall call *"cancellation rule"* or *"*λ*-evaluation rule"*) describes our intention: the inverse nature of abstraction and application.

What does it mean by *substitutable*? Consider $\phi = \lambda(y, \lambda(x, y))$ which is a function that returns a constant-y function on input y. Now, if we perform the substitution

$$\lambda(x, y)[y/x] = \lambda(x, x)$$

which is the identity function. Therefore, we MUST NOT have

$$\phi(x) \vdash \lambda(x, y)[y/x]$$

[7]Bear similar clauses (ii) and (iii) in mind, they introduces the concept of eager versus lazy evaluation in functional programming

because it violates our original intention: $\phi(x)$ is supposed to return a constant-x function. In this case, we say that x is not substitutable for y in ϕ. In general, β is substitutable for x in α if $\alpha[x/\beta]$ does not make any free variables in β become bound.

Let us resume the earlier example:

$$\begin{aligned}\lambda(y,\lambda(x,y))(x) &\vdash \lambda(y,\lambda(z,y))(x)\\ &\vdash \lambda(z,y)[y/x] = \lambda(z,x)\end{aligned}$$

The first line is due to structural rule: namely, we replace $\lambda(x,y)$ by $\lambda(z,y)$ using $\lambda(x,y) \vdash \lambda(z,y)$ (application of alphabetic invariant rule). The second is by cancellation rule.

Remark: A typical strategy to apply the last rule on the pattern $\lambda(x,\alpha)(\beta)$ is to first select a variable y which does not appear at all in both α and β. We use alphabetic invariant rule to get

$$\lambda(x,\alpha) \vdash \lambda(y,\alpha[x/y])$$

and then apply cancellation to obtain

$$\lambda(y,\alpha[x/y])(\beta) \vdash \alpha[x/y][y/\beta]$$

which has a similar effect of directly substituting β for x into α i.e $\alpha[x/\beta]$.

Exercise 1: Show that the choice of y make it substitutable for x in α and then β is substitutable for it in $\alpha[x/y]$.

3. Programming in λ-Calculus

Defining the notion of λ-expressions is already creative. And figuring out the computational capability of such thing requires yet a greater amount of creativity.

Fundamentally, λ-calculus is invented to describe algorithms formally[8]. There are two problems:

(i) How to use λ-expressions to define functions i.e. to write programs?
(ii) Given a λ-expression, what does it really compute?

The second problem is a *very hard* problem concerning *program semantic*. This section will only deal with the first problem. In particular, I will recover the familiar notion of imperative programming such as

- *Primitive data types*: boolean and natural numbers

[8]In other words, the problem it tried to solve is to classify computable functions (such as $F(n)$ that returns the n-th Fibonacci number) from non-computable ones (such as the function $R(n)$ that returns a *random* number)

- *Compound data types*: pair, list (array), structures (i.e. object oriented programming)
- *Control flow constructs*: `if-then-else`, `for`-loop, `while`-loop and general recursion

Before going on, let me make some simplification on the notation. First of all, I shall use

$$x_1, x_2, ..., x_n \mapsto \phi$$

to denote the expression

$$\lambda(v_1, \lambda(v_2, ..., \lambda(v_n, \phi)...)).$$

Secondly, the expression

$$\phi_1\phi_2...\phi_n$$

or the more familiar

$$\phi_1(\phi_2, ..., \phi_n)$$

will also be used in place of the consecutive function application

$$\phi_1(\phi_2)(\phi_3)...(\phi_n).$$

Where do we go from here? We want to use λ-expressions to write functions like `Fib(n)` which returns the n-th Fibonacci number. Recall that each λ-abstraction represents a function mapping $\Lambda \to \Lambda$. So the basic idea is to use λ-abstractions to write functions (i.e. algorithms) and select the appropriate expressions for data structures. In particular, we want to get the expressions $\top$, $\bot$ and a collection of expressions $\{\phi_n\}$ to represents boolean constant **true**, **false** and the natural numbers. Then, *implementation* of a function F, say of form $\mathbb{N} \to \mathbb{N}$ like Fibonacci, is tantamount to constructing an expression ϕ_F of the form[9] $x \mapsto \Box$ such that the application $\phi_F(\phi_n)$ is equivalent to $\phi_{F(n)}$ i.e. $\phi_F(\phi_n) \vdash \phi_{F(n)}$, the expression we used to represent the natural number $F(n)$.

The point is: selection of data representation ($\top$, $\bot$, etc) should allows us to *easily implements* the **operators** on them. For instance, given the choice of each ϕ_n, we might want to implement addition function i.e. $+ : \mathbb{N}, \mathbb{N} \to \mathbb{N}$ with $n, m \mapsto n + m$. Put it literally, we want an expression ϕ_+ such that

$$\phi_+(\phi_n, \phi_m) \vdash \phi_{n+m}$$

for all $n, m \in \mathbb{N}$. If we choose $\phi_n = v_n \in V$ then there is NO way to define ϕ_+.

Exercise 2: Prove that there is really no way. Seriously!

[9] I shall use a $\Box$ at places where an expression is required and should be filled in later.

If you need a hint: look for the *invariant* between equivalent expressions. If $\phi \vdash \psi$ then what is not changed in this transformation? It turns out that their set of their free variables must agree: $FV(\phi) = FV(\psi)$.

If we are able to define such ϕ_+ such that $\phi_+(v_n, v_m) \vdash v_{n+m}$ for all $n, m \in \mathbb{N}$ then we must have $FV(\phi_+(v_n, v_m)) = FV(v_{n+m})$ for all n, m. By definition, $FV(\phi_+(v_n, v_m)) = FV(\phi_+) \cup \{v_n, v_m\}$ while $FV(v_{n+m}) = \{v_{n+m}\}$. Since ϕ_+ is always a finite string, $FV(\phi_+)$ is finite and thus we can pick up a number $k \in \mathbb{N}$ to be the *maximum* index such that $v_k \in FV(\phi_+)$. Apply the invariant property for $n = m = k+1$, we expect $FV(\phi_+) \cup \{v_{k+1}\} = \{v_{2k+2}\}$. This is impossible since $2k + 2 > k + 1 \geq 1$!

Exercise 3: Prove that if one can define some collection of natural number representation i.e. $\{\phi_n : n \in \mathbb{N}\}$ and is able to implement all the operators $\phi_+, \phi_\times, ...$ then one can find a collection of representation in which every expression is free of free variables.

Hint: Instantiate them all.

In general, given the choice of $\{\phi_n : n \in \mathbb{N}\}$, the question whether we can find a definition of ϕ_+ is extremely hard to decide!

3.1. **Boolean logic.** It turns out that the choice of $\top, \bot, \phi_n$ all depends on what operators we want to have on them are. For boolean logic, we will want to define $\phi_\wedge, \phi_\vee, \phi_\neg$ for logical `and`, `or`, `not` operations as well as ϕ_{ite} for the conditional `if-then-else` construct. For natural numbers, we want to be able to define at least ϕ_+ and $\phi_\times$ for addition and multiplication and then possibly further define ϕ_{exp} and ϕ_{iszero} for exponentiation and zero-checking.

Without further ado, let me present *a solution*:

$$\top := x, y \mapsto x = \lambda(x, \lambda(y, x))$$
$$\bot := x, y \mapsto y = \lambda(x, \lambda(y, y))$$

Notice the meaning of the two expressions: the first one is a function that return a constant-x function while the second one is a function that regardless of the input, return the identity function (i.e. it is constant-identity-function function). As the notation conveys, they can also be viewed as function taking two arguments and return either the first or the second input (a *projection* or a *selector*). The following expression

$$\phi_{\mathsf{ite}} := x, y, z \mapsto x(y, z)$$

implements `if-then-else`. Why is that? Think of the application $\phi_{\mathsf{ite}}(\alpha, \beta, \gamma)$ where α, β, γ are arbitrary expressions. If $\alpha \vdash \top$ then

$$\phi_{\mathsf{ite}}(\alpha, \beta, \gamma) \vdash \phi_{\mathsf{ite}}(\top, \beta, \gamma) \vdash \top(\beta, \gamma) \vdash \beta$$

Similarly, if $\alpha \vdash \bot$ then $\phi_{\mathsf{ite}}(\alpha, \beta, \gamma) \vdash \gamma$. In fact, the expressions $\top$ and $\bot$ are chosen as selectors to easily get ϕ_{ite}.

Now, what about logical connectives? The expressions

$$\begin{aligned}\phi_\wedge &:= x, y \mapsto x(y, \bot)\\ \phi_\vee &:= x, y \mapsto x(\top, y)\\ \phi_\neg &:= x \mapsto (y, z \mapsto x(z, y))\end{aligned}$$

implement logical and, or and negation respectively.

How did I get these? Think of the inherent meaning of the function. For instance, we expect $\phi_\neg(\top) \vdash \bot$ and $\phi_\neg(\bot) \vdash \top$. Let's interpret this literally:

- **Input**: the 2-argument-return-the-first function
  Output: the 2-argument-return-the-second function
- **Input**: the 2-argument-return-the-second function
  Output: the 2-argument-return-the-first function

Think more abstractly, we can see that "logical negation" in this case simply *swaps the role of the two arguments* of the supplied function input to $\phi_\neg$ and the definition $y, z \mapsto x(z, y)$ does just that. (The expression $x \mapsto (y, z \mapsto x(y, z))$ is equivalent to identity function on $\top$ and $\bot$!)

3.2. **Natural numbers.** Now, let us tackle natural numbers: choose

$$\phi_n := f, x \mapsto \underbrace{f(f(f...(f}_{n \text{ times}}(x))...)$$

Literally, ϕ_n can be viewed as a two-argument function with one of them (i.e. the first input) being a one-argument function and return the value obtained by applying the one-argument function n times on the second input. This is what I shall call *functional/computational nature* of natural numbers (as opposed to the ordinal/cardinal nature).

Then we get

$$\begin{aligned}\phi_+ &:= x, y \mapsto f, z \mapsto x(f, y(f, z))\\ \phi_\times &:= x, y \mapsto f, z \mapsto x(y(f, z))\\ \phi_{\mathsf{exp}} &:= x, y \mapsto y(\phi_\times(x), \phi_1)\\ \phi_{\mathsf{iszero}} &:= x \mapsto x((y \mapsto \bot), \top)\\ \phi_{\mathsf{iseven}} &:= x \mapsto x(\phi_\neg, \top)\end{aligned}$$

How to get these implementations? Again, think of it functionally. We expect $\phi_+(\phi_n, \phi_m) = \phi_{n+m}$ which literally expand to ϕ_+ taking a function that apply f for n times and a function that applies f for n times and returns a that applies f for $m+n$ times. Evidently, to apply

f for $m+n$ times, we can first apply it for n times, take the result and then apply f to it for m times. From the definition, we should expect

$$\phi_k(f,x) = \underbrace{f(f(...(f}_{k \text{ times}}(x)...)$$

so that

$$\begin{aligned}\phi_n(f,\phi_m(f,x)) &= \underbrace{f(f(...(f}_{n \text{ times}}(\phi_m(f,x))...) \\ &= \underbrace{f(f(...(f}_{n \text{ times}}(\underbrace{f(f(f...(f}_{m \text{ times}}(x))...))...) \\ &= \underbrace{f(f(...(f}_{m+n \text{ times}}(x))...))...)\end{aligned}$$

So the expression $x(f, y(f, z))$ generalizes this expectation by replacing x for ϕ_n and y for ϕ_m gives the implementation. The rests are obtained with similar reasoning.

Exercise 4: Find a way to represent integer.

Exercise 5: Implement more complicated operations on natural numbers such as subtraction and division. If this is not easy or even possible, how should the definition be fixed?

> *Hint:* Implement predecessor $\phi_n \mapsto \phi_{n-1}$ first. Recall that n is supposed to perform $f(f(...f(x)...)$ for n times. Think of the imperative program:

```
y = x
for i = 0 to n-1 do
    y = f(x)
return y
```

> To create $n-1$, we need to return something that is equivalent to $f(f(...f(x)...)$ for $n-1$ times. To do so, we will need to produce a function g from f so that by applying $g(x)$ for n times, you can extract $f(f(...f(x)...)$ ($n-1$ applications)! The solution is to think of g as having another input: a flag to tell whether it has apply f or not as in

```
flag = false; y = x
for i = 0 to n-1 do
    if (flag) then
        y = f(x)
    else
        flag = true
return y
```

> which basically *skips the first application* of f. An alternative is to fix the definition of ϕ_n: represent ϕ_n using a pair $\langle f, x \mapsto f(f(...f(x))), \phi_{n-1}\rangle$. The trade-off is clear: the definition of ϕ_+, $\phi_\times$, ... need to be fixed accordingly and this is hard.

Exercise 6: Define ϕ_{Fib} which computes Fibonacci numbers i.e.

$$\phi_{Fib}(\phi_n) \vdash \phi_{Fib(n)}$$

for all $n \in \mathbb{N}$.

3.3. **Behind the scene.** This section is for the curious: how did I get the magical choices of ϕ_n in the first place? Again, suppose that defining ϕ_+ is our only goal. Clearly, we expect ϕ_+ to be of the form $x, y \mapsto \square$. What are the non-atomic (i.e. not simply x or y) possible expressions formed using two variables x and y? There are infinitely many such expressions, with the *simplest*[10] is perhaps $x(y)$. My goal now is to find some appropriate choices such that the implementation

$$\phi_+ := x, y \mapsto x(y)$$

works. In that case, we have

$$\phi_n(\phi_m) \vdash \phi_{m+n}.$$

This says that the sum is obtained by *applying the number n on the function m.* How is it possible for a *piece of data* to be applied on another piece of data? It means that that piece of data is a function! In other words, ϕ_n should be expected to be a λ-abstraction and that each natural number n, besides their familiar counting role, is *identified* with the $+n$ *function* i.e. the function $g(x) := x + n$. In other words, for the aforementioned definition of ϕ_+ to work, we need to choose ϕ_n to be the $+n$ function!

In particular, ϕ_0 must be a $+0$ function. But for natural number, the $+0$ function is just identity function: $n + 0 = n$. Therefore, we can simply choose

$$\phi_0 := x \mapsto x.$$

I do not know how to choose ϕ_1 but I expect it to be of the form $x \mapsto \square$. Clearly, we cannot fit x to the $\square$ as it will make $\phi_1 = \phi_0$. Again, the simplest one is $x(x)$ so pick

$$\phi_1 := x \mapsto x(x)$$

and see how things go. Now, from the equation $2 = 1 + 1 = +1(1)$ i.e. 2 can be obtained by applying the $+1$ function on the number 1, I suspect that I can just pick

$$\begin{aligned} \phi_2 &:= \phi_1(\phi_1) \\ &\vdash x \mapsto x(x)(\phi_1) \\ &\vdash \phi_1(\phi_1) && \text{(cancellation rule)} \end{aligned}$$

[10]Recall Occam's razor.

One can easily see that no other expression is derivable from ϕ_2! This is **BAD**: the expression is not a λ-abstraction while we need to maintain ϕ_2 to be identified with the $+2$ function. Despite being bad, we can still go on with the logic to obtain $\phi_3 = +1(2) = \phi_1(\phi_2) \vdash \phi_2(\phi_2)$ and in general $\phi_{n+1} = \phi_n(\phi_n)$. The problem is that the intended definition ϕ_+ does not work anymore.

A fix requires a change of mind: if I want ϕ_2 to be a λ-abstraction, why don't I think more functionally, say,

$$+2 = (+1) \circ (+1)$$

i.e. the $+2$ function can be obtained by applying the $+1$ function twice? With this fix, I can derive

$$\begin{aligned} \phi_2 &= x \mapsto \phi_1(\phi_1(x)) \\ &\vdash x \mapsto \phi_1(x(x)) \\ &\vdash x \mapsto x(x)(x(x)) \end{aligned}$$

or more generally

$$\phi_{n+1} = x \mapsto \phi_1(\phi_n(x)).$$

And it should work.

Exercise 7: Check that it do work.

> *Note:* I did not check this formally but there is a good reason for it. Let temporarily denote by φ_+ and φ_n to be our choices of ϕ_+ and ϕ_n in this section and retain ϕ_+ and ϕ_n for the earlier definition. One can verify that $\varphi_n := \phi_n(\varphi_1)$ and $\varphi_+(x, y) := \phi_+(x, y)(\varphi_1)$.

That is about addition. How about multiplication? This choice does not allow me to implement $\phi_\times$ easily. At least, no implementation comes to me immediately. This means, another change of mind is required. This time, I have the basis and experience to figure out the solution: *think functionally*! First of all, in order to make implementation of ϕ_+, $\phi_\times$, ... easy, we need to understand their nature: they are all iterated applications of some operations. Given that I have figured out

$$+n = \underbrace{+1 \circ +1 \circ \cdots \circ +1}_{n \text{ times}}$$

so similarly, we should also notice

$$\times n(m) = \underbrace{+m \circ +m \circ \cdots \circ +m}_{n \text{ times}}.$$

The requirement to plug in different operators suggests to me that I should use some form of higher λ-abstraction such as

$$x, y \mapsto \square$$

instead of the simple $x \mapsto \Box$ so that I easily manipulate the expressions to get *applications of an operation n times.* This is how I figure out the functional nature of natural numbers and get the original definition!

3.4. **Conpound data structures: pairs, tuples and list.** Suppose that α, β are two expressions. Again, we want to represent the ordered pair $\langle \alpha, \beta \rangle$ using the expression $\phi_{\langle \alpha,\beta \rangle}$ in the way that we can later define ϕ_{first} and ϕ_{second} which takes in a pair and returns the first and second component respectively:

$$\phi_{\mathsf{first}}(\phi_{\langle \alpha,\beta \rangle}) \vdash \alpha$$
$$\phi_{\mathsf{second}}(\phi_{\langle \alpha,\beta \rangle}) \vdash \beta$$

Here, the object-oriented programming experience comes to help. Think of $\phi_{\langle \alpha,\beta \rangle}$ as an *object* for which we can supply the methods to access its first and second component. To mimic that, we need to make $\phi_{\langle \alpha,\beta \rangle}$ of the form

$$m \mapsto \Box$$

so that latter we can supply some appropriate expression, say μ_1 and μ_2, to it and obtain the desired result i.e.

$$\phi_{\mathsf{first}} := x \mapsto x(\mu_1)$$
$$\phi_{\mathsf{second}} := x \mapsto x(\mu_2)$$

In other words, we expect

$$\phi_{\mathsf{first}}(\phi_{\langle \alpha,\beta \rangle}) \vdash \phi_{\langle \alpha,\beta \rangle}(\mu_1) \vdash \alpha$$
$$\phi_{\mathsf{second}}(\phi_{\langle \alpha,\beta \rangle}) \vdash \phi_{\langle \alpha,\beta \rangle}(\mu_2) \vdash \beta$$

What do $\phi_{\langle \alpha,\beta \rangle}(\mu_1)$ and $\phi_{\langle \alpha,\beta \rangle}(\mu_2)$ resemble? The applications of boolean values $\top(\alpha, \beta)$ and $\bot(\alpha, \beta)$, of course. In other words, let

$$\mu_1 := \top$$
$$\mu_2 := \bot$$
$$\phi_{\langle \alpha,\beta \rangle} := m \mapsto m(\alpha, \beta)$$

and we get what we want. Generalizing this phenomenon allows us to represent tuples of k expressions (in effect, any class as in OOP) using:

$$\phi_{\langle \alpha_1,\alpha_2,\ldots,\alpha_k \rangle} := m \mapsto m(\alpha_1, \alpha_2, ..., \alpha_k)$$

The philosophy is: representations is developed to work together with methods to obtain information about them.

Exercise 8: Scheme/LISP provides `cons` to construct pairs. Implement it.

Exercise 9: Find a way to represent a list. The list data structure should allow us to easily check whether it is empty, get the head of the list and the tail of the list (i.e. `eq nil`, `car` and `cdr` in Scheme, respectively).

3.5. **Recursion and `while`-loop.** In imperative programming, `while` loop is of the form `while b do c` where `b` is a boolean-valued expression and c is some statement. Unfortunately, this form does not have a direct counterpart in λ-calculus because λ-calculus does not have a notion of *side effect* of a statement. To do so, one need to understand the *"functional role"* of such construct i.e. what the input and the output should be.

Similar to `if-then-else` construct, `while` loop can be viewed as a function which transform the *initial state* (right before the loop begins) to *final state* (right after the loop exits) where state refers to the snapshot of current values of all the program variables. For example: the loop

```
while (n != 0) do n = n + 1;
```

can be identified with the *recursive* function

$$g(n) := \begin{cases} n & \text{if } n = 0 \\ g(n+1) & \text{otherwise.} \end{cases}$$

More generally, the loop

```
while c(s) do s := u(s)
```

can be identified with the function

$$l(s) := \begin{cases} s & \text{if } c(s) \text{ does NOT hold} \\ l(u(s)) & \text{otherwise.} \end{cases}$$

where u has the role of a state updating function and c serves as the contiuation condition. For any initial state s_0, $l(s_0)$ should be the final state.

Our goal of this section is to define an expression ϕ_γ such that $\phi_\gamma(c, u, s_0)$ computes exactly $l(s_0)$. (I purposely choose ϕ_γ because the Greek letter γ looks like the English letter *"Y"* which is a homophone of *"while"*.) It is tempted to literally translate the above definition of $l(s)$ to get ϕ_γ as

$$\phi_\gamma := c, u, s \mapsto \phi_{\mathsf{ite}}(c(s), \underbrace{\phi_\gamma(c, u)}_{l(*)}(u(s)), s)$$

but this does not work! The problem is that: we cannot refer to ϕ_γ. A λ-expression cannot be a sub-expression of itself. Stuck?

Let me bring back the experience of the previous section where we used expressions to represent compound object. If we apply the same paradigm (i.e. *use object to encapsulate computation*), we would like to have γ of the form $c, u \mapsto \square$ so that $\gamma(c, u)$ *is* an object (like a pair/tuple, probably we can call it a *while loop executor*). As an object, $\gamma(c, u)$ has a method (i.e. a constant expression, probably depending of c and u), say μ, so that if we invoke μ on the right parameters, we get the function l:

$$\gamma(c, u)(\mu)(s, \textit{ other appropriate parameters })$$

implements $l(s)$. Sound simple?

Once γ is achieved, the original ϕ_γ can be obtained by defining

$$\phi_\gamma := c, u, s \mapsto \gamma(c, u)(\mu)(s, \textit{ other appropriate parameters }).$$

Now, for this to happen, we expect $\gamma(c, u)$ to be of the form

$$m, s, x, y, ... \mapsto \square.$$

Here, let us recall the earlier temptation i.e. put something similar to

$$\phi_{\mathsf{ite}}(c(s), \phi_\gamma(c, u)(u(s)), s)$$

in the empty box. The difficulty of self-referencing still persists, but this time it is not ϕ_γ but

$$\gamma(c, u)(\mu, u(s), ...)$$

and the wind is on our favor: $\gamma(c, u)$ is an object! In the context of object-oriented programming, if one needs the reference to the object itself in order to implement some method, what should one do? The answer is: one *passes the object as argument to the method.* If one gets this hint, a feasible solution is immediate:

$$\mu := \gamma(c, u) = m, s, x \mapsto \phi_{\mathsf{ite}}(c(s), x(m, u(s), x), s)$$

with the extra x is intended to be instantiated with $\gamma(c, u)$ so that the `then` part of the ϕ_{ite} i.e. $x(m, u(s), x)$ can be interpreted as *invocation of method m of x on extra input $u(s)$*[11]. Putting it all together, we get

$$\phi_\gamma := c, u, s \mapsto \mu(\mu, s, \mu)$$

Exercise 10: What does $\mu(\mu, s, \mu)$ literally mean?

As a remark, m is never used in the above solution. Therefore, one can obtain a simpler alternative with

$$\begin{aligned} \gamma &:= c, u, s, x \mapsto \phi_{\mathsf{ite}}(c(s), x(u(s), x), s) \\ \mu &:= \gamma(c, u) \vdash s, x \mapsto \phi_{\mathsf{ite}}(c(s), x(u(s), x), s) \\ \phi_\gamma &:= c, u, s \mapsto \mu(s, \mu) \end{aligned}$$

[11]Equivalent to `x.m(u(s), x)` in Java

by we identifying the object with the method it provides. (This is analogous to identifying the natural number 1 with the function $(+1)$: $x \mapsto x + 1$ as we did previously.)

Exercise 11: Implement the simple `for i = 0..n` loop.

Exercise 12: I define $\gamma(c, u)$ to construct a `while` loop executor object which has a method to invoke with some s, return $l(s)$. If I had defined $\gamma(c, u, s)$ to mean an object with a method invoking which return exactly $l(s)$, how would the discussion have gone?

Exercise 13: Work out the example in this section. This loop is an infinite-loop if the input value is not zero. What does it mean in λ-calculus?

Exercise 14: Recall the functional role of natural number n is to compute

$$\underbrace{f(f(...f}_{n \text{ times}}(x))...)$$

Then one can ask: what is the corresponding functional role of the other *cardinals*? The goal of this exercise is to find an expression $\phi_{\aleph_0}$ of the form $f \mapsto \Box$ such that $\phi_{\aleph_0}(f)$ *"intuitively computes"*

$$\underbrace{f(f(...f}_{\aleph_0 \text{ times}}(x))...)$$

i.e. the application of $\aleph_0$ (the cardinality of $|\mathbb{N}|$) many times of f.

Hint: What does the loop `while true do s = f(s)` do?

Exercise 15: This exercise is inspired by Cantor's theorem which says that $2^{\aleph_0} > \aleph_0$. We ask the computational analogue of Cantor's theorem: repeated application of a function for $2^{\aleph_0}$ times is NOT *computationally equivalent* to application of the same function for $\aleph_0$ times, does

$$\phi_{\mathsf{exp}}(\phi_2, \phi_{\aleph_0}) \nvdash \phi_{\aleph_0}$$

or the more complicated

$$\phi_{\mathsf{exp}}(\phi_{\aleph_0}, \phi_{\aleph_0}) \nvdash \phi_{\aleph_0}$$

hold where ϕ_{exp} is previously defined to perform exponentiation *for natural numbers* and $\phi_{\aleph_0}$ is as in previous exercise. If it does not hold, what does it mean? Do other theorems concerning cardinal arithmetic, for instance,

$$\phi_+(\phi_{\aleph_0}, \phi_{\aleph_0}) \vdash \phi_{\aleph_0}$$

hold?

Note: I am looking for a soundness-style argument in logic used to prove independence. The confusion is between ordinal and cardinal: $\phi_{\mathsf{exp}}(\phi_2, \phi_{\aleph_0})$ does not apply f for $2^{\aleph_0}$ times! This exercise also conveys one message: no λ-expression can return an uncountable ordinal. I define a *ordinal* to be computable if it can be computed by a λ-expression.

Exercise 16: Write a program to evaluate λ-expressions i.e. recursively apply the rules in definition 3 until nothing else is derivable. Use this program to check the solutions of other exercises.

Exercise 17: Forget everything and rewrite everything from scratch.

4. Conclusion

After a laborious mental exercise, it is time to end the article with a

Definition 4. *A function* $F : \mathbb{N} \to \mathbb{N}$ *is* computable *if and only if it can be implemented using a* λ*-expression i.e.* λ*-definable.*

The thesis of this work is, as in the introduction, λ*-calculus is the computational analogue of first-order logic.* And there is still much more to explore.

To C. B.