A Programmer's View of Trusting Programs on Pure Mathematics

Manabu Sumioka

1. Prologue

I thank Matsuyama University for the network accessibility from the early Internet era, especially not only for teachers but also for all students who want to have access to the Internet. They can enjoy the web resources very much depending on their interest and their ability of handling a computer. Although now the university's programming course is less weighted in the curriculum.

But I still have to insist that programming experiences is very important for any people to use a computer even if they use computer just for writing reports or calculating some accounts after for their rest of the life.

Think, simply think sitting before a computer and browsing Web pages as a novice computer user. A browser will be used for the purpose. You may wonder how the browser display the pages on the screen of your

computer. It depends on the program that how it render the HTML text which get from a Web server.

So as to show you writing a short program is not so difficult for any computer user who have enough knowledge about the domain where he/she is willing to do some tasks and not so difficult to understand the program, I will show some code example (figure. 1) written in Python (http : //www. python. org) which get the raw HTML from a Web server and print the results. This program code uses Python httplib module.

Figure 1 A python httplib code fragment but it works for getting a raw HTML from a Web.

```
import httplib
h = httplib. HTTP ('www')
h. putrequest ('GET', '/index. html')
h. putheader ('Accept', 'text/html')
h. putheader ('Accept', 'text/plain')
h. endheaders ( )
errcode, errmsg, headers = h. getreply ( )
f = h. getfile ( )
data = f. read ( ) # Get the raw HTML
f. close ( )
print data
```

A Web browser program start from this point that to get the raw HTML. Any program which seems to do some complex task start from such a simple code fragment and as Eric S. Raymond said in 'The Cathedral and the Bazaar' that it's almost easier to start from a good partial solution

than from nothing at all. Raymond founded an idea of open-source that depends on some theories about software engineering suggested by the history of Linux.

As a programmer, I write program. My occupation and programming experience started on old UNIVAC 1100 machines. Quoting some codes which I wrote and referred to I would like to present programming is how cute in some way for you.

2. Euclid's algorithm and two way programming style

As D. E. Knuth states in his famous text book "The Art of Computer Programming volume 1", basis of all of computer programming is "Algorithm". The word algorithm starts from Euclid's algorithm, a process to find the GCD (greatest common divisor) of two positive integers.

Euclid's algorithm is:

(1) Divide X by Y and let R be the remainder.

(2) IF R=0, the algorithm terminates; X is the answer.

(3) Set Y to X, R to Y, and go to (1).

The algorithm consists of 3 steps, it starts from step (1). Each step of the algorithm do simple arithmetical calculation and after some arithmetical decision go to the specified step. The control goes to next step if "go to" or "terminate" is not specified.

Above Euclid's algorithm does not exhibit same as described in original Euclid's book but this somehow formalized description comes from the procedural computation form. This is one formalism of computation. On python programming language system, we can write the following program which represents Euclid's algorithm.

Figure 2 A python program for Euclid's algorithm

```
r = x % y
While r != 0:
        x, y = y, r; r = x % y
print x
```

This is a procedural program presented on the imperative computation model. The imperative computation model stands on von Neumann machine architecture. We can show another program which comes from another computation model.

According to a formal system defined by Herbrand-Gödel-Kleene (HGK), we can define GCD as a set of equations on natural numbers. The HGK system is some type of functional programming system which uses substitutions and deductions formalized by Kleene.

Figure 3 A GCD program for the Herbrand-Gödel-Kleene formal system

Equation 1. gcd (X, 0) = X

Equation 2. gcd (X, Y) = gcd (Y, remainder (X, Y))

In equation 2, remainder (X, Y) is used as an already defined function for HGK to calculate the remainder after dividing X by Y. For the calculation of GCD (X, Y), we will define a functional (or schema) G for HGK as follows:

G (gcd, X, Y) ≡ if Y=0 the X else G (gcd, Y, remainder (X, Y))

The functional G (gcd, X, Y) is an auxiliary function.

A GCD program for HGK will be presented for 'Lisp' using the functional G (GCD, X, Y) as g (f x y) in figure 4 lisp program. Specifically, the function 'g' is defined using 'g' itself as a first argument. Self-application likeness is allowed in Lisp programming language.

Figure 4 A lisp program for GCD

```
(defun gcd (x y)
  (g (function g) x y))
(defun g (f x y)
  (cond ((zerop y) x)
        (t (g f y (rem x y)))))
```

Lisp is a functional programming language. In an imperative language like Python the question is "In order to do something what operations must be carried out, an in what order?". In functional programming language the question is "How can this function be defined?". Above Lisp program defines the function GCD. In Lisp functions are partial recursive functions. "Partial" means in some case they may not have a value.

3. Code as a mapping domain

Everybody including computer beginner to advanced computer user must use a keyboard as a communication device for computers. Roughly speaking, our idea to be executed on computer should be described as strings in some set of symbols at last. The computing mechanism of von Neumann machines come from Turing machines. Turing machines are some kind of machines each which is designed to reproduce all sorts of operations which a human could do. For any function which is effectively calculable, a Turing machine can be found which computes it. This is formulated as Turing's thesis (received for publication 28 May 1936) that every function which would be naturally be regarded as computable is computable under one of his machines.

In a mathematical theory, formalization of a theory did an important

role particularly. Ancient mathematician Pythagoras and Euclid discovered the axiomatic deductive method in mathematics. At the beginning of 20th century, Hilbert emphasized that strict formalization of a mathematical theory called formal system or formal theory. His method of making the formal system is called metamathematics which treat as a whole of object of a mathematical study.

We define two terms, "object theory" and "metatheory" in a particular formal theory. An object theory relates to the system itself and a metatheory relates to the metamathematics relating to the formal theory. We suppose **L** as an object theory and **M** for a metatheory. Also we can call **L** as the object language and **M** as a meta language.

Proofs in **L** are formalized in **M** according to the methods invented by Gödel. It starts from the arithmetization of metamathematics or metamathematics as a generalized arithmetic. It was illustrated in the theorem "on formally undecidable propositions of Principia Mathematica and related systems" by Gödel.

All objects in a formal system is represented by strings of finite length. So the formal objects are enumerable. Therefore the whole of the objects in **L** has one to one correspondence to the whole natural numbers set **N**, and we can formulate as

$$\mathbf{G}: \mathbf{L} \rightarrow \mathbf{N},\ \boldsymbol{x} \rightarrow \ulcorner \boldsymbol{x} \urcorner$$

Where $\boldsymbol{x}$ is any object in **L** and $\ulcorner \boldsymbol{x} \urcorner \in \mathbf{N}$ is a Gödel number of the object $\boldsymbol{x}$. We can define that **G** is a primitive recursive function and object $\boldsymbol{x}$ is decodable from $\ulcorner \boldsymbol{x} \urcorner$. According to **G**, embedding the formal theory into the number theory, the arithmetization of metamathematics is done. The concept of a Gödel numbering is a natural way of expression for computer programmers who are familiar to express objects as bits strings (are natural

number). I also believe that von Neumann invented the computer architecture from this Gödel numbering idea.

The arithmetization of metamathematics succeeded in discussing the idea dealing with meta or higher order concept. By an introduction of the arithmetization of metamathematics, we can deal the meta concept and the original concept at the same level.

Some famous paradoxes were founded at the beginning of 20th century.

$A \Leftrightarrow$ non-A (Richard paradox 1905)

$x \in x \Leftrightarrow x \notin x$ (Russell paradox 1902-3)

The Richard paradox deals with the notion of finite definability. He gave the paradox in a form relating to a real number and a more popularly stated one in English is "the least natural number not nameable in fewer than twenty-two syllables". This expression names in twenty-one syllables a natural number which by definition cannot be named in fewer than twenty-two syllables. Therefore, it is a paradox.

Another famous paradox called the Russel paradox exists which comes from Cantor's set theory. The paradox deals with the set of all sets are not members of themselves. A popularization of the paradox concerns the barber in a village, who shaves all and only those persons in the village who do not shave themselves. Does he shave himself ?

Generally speaking, applying a function to itself f (f) is called "self -application". We will show some paradox occurs in the self-application world. We assume that **V** is a set which contains at least two element and

For all f, $x \in \mathbf{V} \Rightarrow f(x) \in \mathbf{V}$

When we defined a function paradox$\in \mathbf{V}$ as follows:

paradox (f) $\equiv$ if f (f) $=$g then h else g

where h and g are different functions in **V**.

If we make an self-application for the function paradox$\in$**V**, we have

paradox (paradox) = g iff paradox (paradox) = h

where iff means that if and only if.

But we made the assumption that $g \neq h$, so this is a case of Russel paradox. Therefore, we have a Russel paradox in the world where functional self-application is allowed.

As we have already described, the computing mechanism of von Neumann machine comes from Turing machines. The storage of machine holds the binary data and instructions which are functions they transform input data to output data, and they are represented as binary strings. There are no differences between data and program those are stored in the memory of computer in a point of view that they are represented as binary digits strings. When those binary strings are interpreted as sequences of instructions, they work as functions which transforms binary digits strings in the memory of computer.

We suppose **M** is set of the memory states and **B** is set of the bit strings in the computer. A function "contents" maps **M** to **B**:

contents: $\mathbf{M} \rightarrow \mathbf{B}$.

For $m \in \mathbf{M}$ and $b \in \mathbf{B}$, the function bits transforms m to b defined as b =contents (m). Set of instructions in the memory of computer is defined as follows:

instruction: $(\mathbf{M} \rightarrow \mathbf{B}) \rightarrow (\mathbf{M} \rightarrow \mathbf{B})$.

If contents (m) is an instruction, the instruction contents (m) is a function which maps current state "contents" to the next state after the execution of the instruction which is represented as:

contents (m) (contents).

Since the function contents do self application.

4. The meta language ML and recursive functions

The meta language ML comes from LCF and Milner's polymorphic type system. A useful information resource about the ML includes how to get ML compilers is in the FAQ (http : //www. cis. ohio-state. edu/hypertext/faq/usenet/meta-lang-faq/faq. html). The formal logic of LCF is a blend of the predicate calculus with equality and the lambda calculus. It was based upon Dana Scott's theory of domains of continuous functions, and is particularly suited to the formulation and proof of properties of algorithm and algorithmic languages. ML applies to the whole area of symbolic computation and list processing is like Lisp. ML supports the functional programming including higher order functions and moreover its supports imperative programming.

A factorial program "fact" for ML is as follows :

```
-fun fact 0 = 1
= | fact n = n*fact (n-1);
val fact = fn: int -> int
```

At the last line ML do an inference that fact is a function which maps integer to integer. Next example is a definition of higher order function "map" applies fact function to each element of a list.

```
-fun map (f, l) =
= if l = nil then nil
= else f (hd l) : : map (f, (tl l));
val map = fn: ('a -> 'b) *'a list -> 'b list
-map (fact, [1, 2, 3, 4, 5] );
val it = [1, 2, 6, 24, 120] : int list
```

Like a Lisp program using functional G (gcd, X, Y), GCD program in

ML version is as follows.

```
-fun g (f, x, 0) = x
= |  g (f, x, y) = g (f, y, x mod y);
val g = fn : 'a * int * int -> int
-fun gcd (x, y) = g (gcd, x, y);
val gcd = fn : int * int -> int
-gcd (16107, 8437);
val it = 767 : int
```

The function twice

(twice (f)) (x) = f (f (x))

works almost like self-application function is given in ML as follows. Is applies f twice to x.

```
-fun twice f x = f (f (x));
val twice = fn : ('a -> 'a) -> 'a -> 'a
```

In ML, 'a means a general type in place of the specific type such as int or string. Since the function twice has type (any type -> any type) -> (any type -> any type). In the real ML, the type of twice is returned as fn : ('a -> 'a) -> 'a -> 'a, but it means ('a -> 'a) -> ('a -> 'a) because the arrow is right associative. The twice is a function which takes a type (any type -> any type) function and returns a function (any type -> any type). We will apply the function twice to a function square which square the integer and see the results.

```
-fun square x = x * x;
val square = fn : int -> int
```

The function twice for square is tested in ML as follows.

```
-square 2;
val it = 4 : int
```

```
-twice square 2;
val it = 16: int
-twice twice square 2;
val it = 65536: int
```

Twice twice is a functional that applies its arguments four times. But this is not really self application in the meta language ML. We will explain why the twice in ML is not a self-application program. We have to start from a expression $\lambda x.\ t$ where x and t are variables and is called lambda expression. Lambda expression is composed of constants, variables, abstractions, and combinations, they are sometimes called λ-term. The syntax of the λ-calculus is simple. Lambda abstraction is the form $(\lambda x.\ t)$ and combination is (t u) which means application of t to the argument u. The lambda expression $\lambda x.\ t$ can be regarded as a function in ordinary mathematics. Lambda abstraction allows us to express a function as $(\lambda x.\ ax^3+bx^2+c)$ without giving a name to it. In an ordinary mathematical expression, we write it as $f(x) = ax^3+bx^2+c$ and give a name f to the function as follows:

$$f:\ x \rightarrow ax^3+bx^2+c.$$

To avoid giving a name, λ is used as an auxiliary symbol by Church and he wrote

$$f = \lambda x.\ ax^3+bx^2+c.$$

But it cause to be involved in higher order functions and semantic questions occurs which we will show later. To explain the reason why the twice in ML is not self application, we will define twice in the λ-calculus. In the λ-calculus, twice is defined by a lambda expression twice $\equiv \lambda f\ x.\ f\ (f\ (x))$. The test of twice in ML is checked by β-conversion in the λ-calculus:

$$(\lambda f\ x.\ f\ (f\ (x)))\ \text{square}\ 2 \equiv (\lambda x.\ \text{square}\ (\text{square}\ (x)))\ 2$$

$\equiv$ square (square (2))

$\equiv$ 16

Because twice f x $\equiv$ f (f (x)) for terms f, x, the expression twice twice square 2 is

twice twice square 2 $\equiv$ twice (twice square) 2

$\equiv$ twice square (twice square 2)

$\equiv$ twice square (square (square (2)))

$\equiv$ square (square (square (square (2))))

$\equiv$ 65536.

For the test of twice, we have the same result in the λ-calculus as in the meta language ML. But we know the function twice is applied to itself. The function twice in ML has type such as:

twice : integer $\rightarrow$ integer.

In twice twice, the first twice and the second twice have different type. The second twice has twice (integer $\rightarrow$ integer) $\rightarrow$ (integer $\rightarrow$ integer). The first occurrence of twice has type ((integer $\rightarrow$ integer) $\rightarrow$ (integer $\rightarrow$ integer)) $\rightarrow$ ((integer (integer) $\rightarrow$ (integer $\rightarrow$ integer)). The first twice and the second twice have different types. Since the different occurrence of twice takes different terms, it is not really self application in ML.

5. Recursive function and Y for ML

The function twice can be applied to itself. Since it is not a conventional function. Let see that a function f (x) =y maps an element x of a set 【x】 to an element of a set 【y】. For the function twice f x $\equiv$ f (f (x)), twice is a set of all pairs (twice, y) such that twice (twice) = y. Thus twice is an element of 【twice】 for some y and it violates the axiom that no set can belong to itself.

Since self application causes Russell's paradox, the formal set theory excludes the self application. But we described that the computational model of von Neumann machine contains self application of the function contents in the last section. The computer program is viewed as rules which transforms input to output. λ-calculus is resembles to the program because it views functions as rules.

Until now we referred λ-calculus as untyped (type free) λ-calculus where the terms are untyped. The terms in the theory λ denoted by the untyped λ-calculus are build up from variables and abstraction.

Consider the factorial function again as the following recursive definition:

FACT $\equiv$ λn. IF (= n 0) 1 (* n (FACT (− n 1)))

Where IF is a conditional function whose behavior is defined by the following reductions:

IF TRUE P R $\rightarrow$ P

IF FALSE P R $\rightarrow$ R.

If n $\geq$ 0 then this definition gives a value n * ... * 2 * 1 for FACT (n). If n $<$ 0 then the evaluation of FACT (n) will not stop. Above recursive definition can be expressed as follows focusing on the recursive structure.

FACT $\equiv$ λn. (... FACT...)

By performing β-abstraction on FACT, it becomes as:

FACT $\equiv$ λfact. (λn. (... fact...)) FACT

This is represented as:

FACT $\equiv$ H FACT

Where

H $\equiv$ λfact. (λn. (... fact...))

FACT $\equiv$ H FACT means that when the function H is applied to FACT,

the result is FACT. FACT is said as a fixed point of H. When H is viewed as a operator, FACT is a fixed point of the operator H. Another example of a fixed point for the recursive function GCD is represented as:

GCD ≡ G GCD

Where GCD is a fixed point of the function G. A function may have more than one fixed point. To seek a fixed point of H, we introduce a function Y which takes a function and gives a fixed of the function as its result. Thus for the fixed point of H, Y behaves as follows:

Y H ≡ H (Y H)

By assuming Y hold above equation, we can give a non-recursive definition of FACT, namely

FACT ≡ Y H

H ≡ λfact. (λn. IF (= n 0) 1 (∗ n (fact (−n 1)))).

Above equation shows that the recursive definition of FACT is turned into a non-recursive definition a functional H.

H (g) (n) ≡ λfact. (λn. IF (=n 0) 1 (∗n (fact (−n 1)))) (g) (n)

As a well known result, Y can be defined as a λ abstraction as follows:

Y ≡ λf. (λx. f (x x)) (λx. f (x x)).

To see Y H ≡ H (Y H), let us evaluate Y H.

Y H = (λf. (λx. f (x x)) (λx. f (x x))) H

= (λx. H(x x)) (λx. H (x x))

= H ((λx. H(x x)) (λx. H (x x)))

= H (Y H)

Third line of above reduction comes from β-conversion. Define FACT ≡ Y H. Then we have FACT satisfying its recursion equation.

(FACT n) ≡ ((Y H) n)

≡ ((H (Y H)) n)

$\equiv$ ((H FACT) n)

$\equiv$ (λn. IF (=n 0) 1 (*n (FACT (−n1))))

How do I write the Y combinator in ML without using a recursive definition? We can give an answer as follows:

```
datatype 'a t = T of 'a t -> 'a
val y = fn f => (fn (T x) => (f (fn a => x (T x) a)))
                (T (fn (T x) => (f (fn a => x (T x) a))))
```

6. Epilogue

The origin of the programming goes back to the beginning of the 20th century. In the year 1900, David Hilbert did a famous lecture which is a call to mathematicians to solve a list of twenty-three difficult problems. His idea is the culmination of two thousand years of mathematical tradition going back to Euclid's axiomatic treatment of geometry, going back to Leibniz's dream of a symbolic logic and Russell and Whitehead's monumental Principia Mathematica. He wanted to formulate a formal axiomatic system which would encompass all of mathematics.

Mathematical ideas helped programmers to make more trusting programs. One of those idea was the λ-calculus. The λ-calculus possesses some features of programming language such that type free aspect which corresponds to a program and data are same. Moreover a program can apply to itself regarding the program as its data, so a program can transform itself. Some programming language feature is also inspired by λ-calculus. Some imperative programming language have a feature that procedures can be argument of procedures. Functional programming language have same feature and moreover procedures can produce procedures as output. There was a need for expressing the functional meaning of a

program, denotational semantics of programming languages. As described in section 3, the self applicable function twice violates the axiom of the set theory. With C. Strachey's work in the denotational semantics of programming languages, Scott developed a LCF in 1969. The meaning of expressions in programming languages can be taken as elements of certain spaces of 'partial' objects. Scott had shown that these spaces are modeled in one universal domain P-omega, the set of all subsets of the integers. This domain renders the connection of this semantic theory with the ordinary theory of number theoretic functions clear and straightforward.

References

1. Eric S. Raymond, The Cathedral and the Bazaar, Nov. 1998. http: //www. tuxedo. org/~esr/writings/cathedral-bazaar/cathedral-bazaar. html
2. Stephen C. Kleene, Introduction to Metamathematics, North-Holland, 1952.
3. John. McCARTHY, Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I, CACM, April 1960. Also available online at http: //www -formal. stanford. edu/jmc/recursive. html
4. Dana Scott, Data Types as Lattices, Oxford University Programming Research Group Technical Monographs, September 1976.
5. Manabu Sumioka, Data types in Scott's D model - Part I. Matsuyama University 40th memorial ronbunshu (Dec. 1990), 453-466.
6. Lawrence C. Paulson, Logic and Computation, Cambridge University Press, 1987.