
The proof-program correspondence a capital discovery ... ignored

Jean-Louis Krivine

IRIF, Université Paris-Cité

krivine@irif.fr

Paris, May 19, 2025

Introduction

In the 1930s, two strange logical objects were discovered:

combinatory logic by the Russian Moïse Schönfinkel

lambda-calculus by the American Alonzo Church.

Their true nature would only be understood 30 years later:

they were *programming languages*!

In the 1960s, independently and ten years apart

H. Curry and W. Howard linked them to a rudimentary system of proofs:

intuitionistic propositional logic

whose expressive power is very weak.

Introduction

This correspondence between proofs and programs
will gradually be extended
to a universal system: *classical set theory*
which allows the expression of all mathematical proofs.

It's a very interesting story
but it would be necessary to go into some 'technical details'.
I want to talk instead about the consequences of this extraordinary discovery
whose impact was almost nonexistent.

The correspondence

Simple statement:

Mathematical proofs are computer programs.

Not vice versa. Not all programs are proofs, far from it.

Precise statement:

Formalized mathematical proofs
are computer programs *in machine language*.

But I will need to explain the two terms *formalized* and *machine language*.

The simple statement will suffice for now
especially regarding applications in theoretical physics.

Immediate consequence

Here's a trivial argument: a syllogism!

Mathematical proofs are computer programs.

Archimedes wrote mathematical proofs.

So Archimedes wrote computer programs.

I deduce that he had a computer: his brain?

No, Archimedes' brain *isn't* a computer.

But it does *contain* one, among a thousand other things.

Introspection

These programs, Archimedes, Euclid, Euler, Gauss, ...
did not invent them. They read them in their heads.

Remember Plato's famous text
in which Socrates *reminds* a slave

(what democracy!)

that he *knows* that the diagonal of a square is equal to $\sqrt{2}$.

But what are all these programs doing in our heads?

And in the head of a zebra or a bat?

Evolution

So does a zebra 'know' that the diagonal of a square is equal to $\sqrt{2}$?

In any case, it knows how to use it, and fortunately for it.

Because it's essential for its survival.

For the same reason, and like you and me,

it knows how to use the law of gravitation,

Maxwell's equations, and quantum mechanics.

But *exclusively* for its own personal use.

Unlike *homo sapiens*, who *also* use them in industry.

Importance of this discovery

The areas of application for this correspondence are fascinating:

- the laws of physics
- the nature of mathematics
- how the brains of advanced animals work

It is currently used to write proven programmes.

That's good, but ... there's room for improvement!

So let's first take a look at *the laws of physics*.

The laws of physics

The so-called *mathematical laws of nature*:
gravity, electromagnetism, relativity, quantum mechanics, etc.
are *programs* written by evolution (**Mother Programmer**)
into the brains of animals to help them reproduce.

The 'unreasonable effectiveness' of mathematics
is the incredible effectiveness of Mother Programmer.

Yet we use these same programs
in completely different conditions
for experimental or industrial purposes.

The Outlaw Universe

We tend to imagine that the Universe follows laws,
which we discover through experiments or observations.

No, the Universe has no laws!

We are right to use our intuition; it is a good research tool.
Biologists have no qualms about talking
about the emergence of an organ because of its usefulness.
This is false, they know it, but it is very convenient.
The laws of the Universe are convenient and just as false.

Mother Programmer

Biologists think about evolution all the time;
it is their fundamental tool.

Why do physicists never think about it?

Not to mention mathematicians.

They are wrong, as we shall see.

And philosophers?

Consequences

This results in three principles that *physicists have never observed*:

No. 1. *In an experiment, the results are intended for the observer.*

And no one else! The observer plays the central role, *whether he likes it or not*.

No. 2. *If the observer is not alone, the laws may be false.*

Obviously, Mother Programmer only deals with one brain at a time!

No. 3. *She does not have unlimited computing power.*

It is the brain of a bat or of a wildebeest's.

We can therefore expect blockages.

Black matter? Dark energy?? The Theory of Everything???

The Observer in Relativity

The dimensions of a body and the duration of an event depend on the observer.

Not surprisingly, this data is reserved for her personally.

It was not intended that a bat and a wildebeest should exchange the results of their measurements.

The two twins are no longer the same age?

Obviously, they just don't have to compare them (principle No. 2).

Paradoxes in quantum mechanics

They disappear if we apply the principles already stated.

1. Information that travels faster than light.

Instantly, in fact (Aspect's experiment).

Obviously, *two observers* are required. This contradicts principle No. 2.

The usual explanation is to redefine the concept of information so that what is transmitted is not information.

Ad hoc: so a cryptographic key is not information?

Ask the hackers.

Yet it is transmitted instantaneously.

Paradoxes in quantum mechanics

2. The wave function's collapse.

An old chestnut that has given rise to some mind-boggling explanations, such as *multiple worlds* appearing by the millions as soon as a physicist (or an ant) conducts an experiment!

All this to avoid applying principle No. 1:

The results of an experiment are intended exclusively for the observer.

Collapse

The *wave function* of an electron gives, at each point, the probability that it will be found there.

It continuously follows a PDE: *Schrödinger's equation*.

Until the moment when the observer *decides* to know its exact position.

The probability then *suddenly* becomes a certainty, and Schrödinger's equation quietly resumes with this new data as its initial condition.

Collapse (the end)

Question :

How did this devilish electron realise that someone had taken a measurement?

Answer:

The electron is a convenient fiction that describes an *interactive program* designed to help the observer act at the right moment and controlled by him.

Moral and thought experiment:

To interpret an experiment correctly, the observer must believe that *his life depends on the result*.

Young's slits

Each so-called 'paradox' in physics is therefore, in fact, a very interesting tool for studying how the observer's brain works.

A bit like optical illusions for vision.

Thus, *Young's double-slit* experiment is particularly relevant.

A striking example of the *uncertainty principle*.

It was in reference to this experiment that Richard Feynman wrote this curious remark:

If you think you understand quantum mechanics, you don't understand quantum mechanics.

Let's take a look!

Young's slits (continued)

The equipment:

An electron source

A screen with two slits

A receiving screen.

The observer prepares to send *a single* electron.

What does quantum mechanics predict?

In general, only probabilities, rarely a definite result.

However, there is an area of the receiving screen that has surprising properties.

The myth of Orpheus

If the observer turns his back during the experiment,
the electron will certainly not hit the area in question.
But if the observer watches which slit the electron passes through,
it has a positive (calculable) probability of hitting it,
and Eurydice returns to Hades.

The moral of the story (stubbornly rejected, despite the evidence):

In an experiment, it is the observer who plays the leading role.

No wonder he is the object of all **Mother Programmer's** attention.

And of course, he must be *unique*.

Otherwise, one might be watching and the other not,
and the electron would not know which way to turn!

Young's slits (continued)

So how does this program work in a brain?

Simply by asking its superior:

Do you have a way of knowing which slit the electron will pass through?

and providing it with the probability based solely on its answer

yes or *no*

without worrying about the nature or even the reality of this means.

What a fascinating research topic

this dialogue between two parts of the brain is:

the application that asks the question and the central system that answers!

Young's slits (the end)

I don't know under what circumstances
this dialogue takes place in the brain of a zebra or an ostrich.

But it must be frequent and vital
for **Mother Programmer** to have been concerned about it.

What if the central system makes a mistake in its binary response?

Then too bad for the zebra!

So it doesn't make mistakes often because there are still zebras.

(Not for much longer, though).

Formalization and compilation

Let's return to the precise statement of the proof-program correspondence:

*Formalized mathematical proofs
are computer programs in machine language.*

The transition from a *readable* proof to a *formalized* proof
is called *formalization*.

The transition from a *readable* program to a *machine-language* program
is called *compilation*.

There are, in fact, two very distinct kinds of languages
whether for proofs or for programs:

Readable and **Unreadable**.

Formalization

So what is a *formal proof*?

At the beginning of the 20th century, logicians discovered:

- all the rules of mathematical language *syntax*
- all the rules of *proof*

and they are surprisingly simple.

These results were very important, but not well received at the time by 'normal' mathematicians (see below).

Syntax

The symbols are: $\perp$ (*false*), $\rightarrow$ (*implies*), $\forall$ (*for all*), $\in$ (*belongs to*), the letters $a, b, \dots, x, y, z$, which are called *variables* without forgetting the two parentheses.

That's it!

We then construct sentences called *formulas*. Example:

$$\forall x \forall y (\forall z (z \in x \rightarrow z \in y) \rightarrow (\forall z (z \in y \rightarrow z \in x) \rightarrow \forall z (x \in z \rightarrow y \in z)))$$

Note: This formula is called *axiom of extensionality*.

Axioms and logical rules

They can be counted on the fingers of one hand and are surprisingly simple.

Examples:

Modus ponens: from A and $A \rightarrow B$, deduce B .

Peirce's law: deduce $((A \rightarrow B) \rightarrow A) \rightarrow A$

also called *excluded middle* or *reasoning by contradiction*.

Elimination of $\forall$: deduce $\forall x A[x] \rightarrow A[y]$.

Note: The famous *syllogism* is formed from an elimination of $\forall$ followed by two modus ponens.

ZF

To complete this formidable task, it was necessary to describe the objects to which this language would be applied: **sets**.

This is the theory invented by Cantor and finally stated in a few axioms by Zermelo-Fraenkel.

I wrote the *axiom of extensionality* above.

Another example is the *axiom of the pair*:

$$\forall x \forall y (\forall z (x \in z \rightarrow (y \in z \rightarrow \perp)) \rightarrow \perp)$$

Formal proofs

Any demonstration can be translated into this language.

This transforms texts that are readable (by specialists, of course) into excessively long and completely unreadable texts.

This is the process of *formalization*.

We understand the mockery and disinterest of 'normal' mathematicians.

But the advent of computer science changed everything.

Computer languages

In computing, there are also two types of languages:

- *High-level languages* such as C, Java, Python, etc.

These are used by *programmers*.

- *Machine languages* written using only

binary characters: 0,1

or, equivalently, but more conveniently,

hexadecimal characters: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

They can only be read by *processors*.

DNA is also written in machine language using the characters A,C,G,T.

No advanced language is known for it.

Compilation

Translating a high-level language program into machine language is called *compilation*.

It's an impossible task for a human.

It can be done on a computer using a suitable program called a *compiler*

and written, of course, in high-level language.

Riddle: How was the compiler compiled?

Compilation

Here is a tiny program in C
that writes 'Hello World!' on the screen :

```
#include <stdio.h>
int main(){
    printf("Hello, World!\n");
}
```

and a very short excerpt from a compiled 'hexa' :

A8 EB 47 68 2C B0 47 50 55 9A 55 F4 76 69 A0 1A DD 8F 51 E3 D5 F9 7E CC 94 48 BF
E9 7C A2 37 C6 F2 98 D6 A8 3D 03 60 73 82 38 58 C5 50 40 D9 6F 76 F2 44 AF 85 E0
4F 71 71 E3 84 F9 59 1C 64 84 59 E4 95 62 B2 D4 B6 E4 38 73 76 23 5C 32 17 E7 66
04 16 E8 25 0E 85 4A EA E7 D4 62 28 7F 72 A9 B3 25 E5 AF 4D 9B 37 D0 98 4A 5B 91
B0 E9 17 A2 C3 4F 1F D6 CB 30 DA 42 04 41 78 8F 00 00 00 00 3A D8 29 67 00 00 00
00 02 00 00 00 1C 01 00 00 58 37 00 00 58 19 00 00 52 53 44 53 E9 E9 69 4B D9 D8
3F 49 B2 EB 14 E7 58 86 A8 3C 01 00 00 00 43 3A 5C 6A 6E 6C 70 5C 77 6F 72 6B 73
70 61 63 6D 27 46 A4 A6 1A 5D F7 54 51 6A 80 22 C6 A8 66 75 4D 54 83 C9 5E 68 1B
AB F6 5B 64 B0 AD AD 20 BD 54 EF 46 6B 6D 02 E7 45 AE 75 F6 8E 8A B9 32 2E 8B B8
77 1C 52 4D 79 88 5F A5 3A 18 C9 50 2C 97 27 2D BF DA 00 7D 63 7F EF 32 EC 46 87
63 DB 9A 9F C3 5A 21 36 7B F9 9B D5 CA A0 6E 72 2C FC E1 F1 9E 70 4D 08 56 15 6F
90 AB FB 48 BA B9 D7 C1 82 78 85 65 2E 7F B5 C9 C0 E2 14 B7 37 11 41 06 E2 8A E0
A8 F6 E5 7E B6 78 05 01 C3 60 ED 4F 07 41 2E 0A 13 49 E3 90 9B 7F DD BD FA AC BF

Decompilation

In industry, we may be faced with the opposite problem:
converting from machine language to high-level language
for maintenance or technical progress reasons.

This is also a key activity for malicious hackers.

This is called *decompilation*. It is always very difficult
and often impossible.

It cannot be programmed.

The correspondence

The Curry-Howard correspondence identifies:

the formal language

of mathematics

formalization

formalized proofs

the machine language

of computer science

compilation

compiled programs.

We will therefore also identify

the usual mathematical language with *a high-level computer language*

such as C, Python, or Java.

Decompilation (continued)

The mathematician's job therefore consists of decompiling the machine language programs in his head into ordinary mathematical language.

An arduous task that has lasted for centuries that no other animal has been able to perform.

These programs, installed and perfected by **Mother Programmer** to enable the survival and reproduction of evolved animals have proven to be fantastically effective.

And now, we irresistibly think of *The Sorcerer's Apprentice*.

Recursive functions

What kind of program do we get from a proof?

Can it be executed, and what happens then?

Let's start with the mathematics of physics,
which are essentially PDEs and probabilities.

The associated programs are the calculations
performed in research centers and industry.

Such a program waits for *numerical* data, runs for a certain time,
then stops with a *numerical* result.

In other words, these programs calculate *recursive functions*.

Systems and networks

But not all programs are like this:

An *operating system* is a program that does not wait for any data and must never stop. It keeps the computer running; any shutdown would be fatal.

Similarly, the programs that run networks work continuously in *routers, servers* and *clients* (including you).

So what branch of mathematics produces such programs?

Set theory

A fascinating research topic:

Writing and running programs associated with ZF theorems would give access to the computer operating system included in the brains of zebras (and Archimedes).

The Curry-Howard correspondence provides us with the tools.

Let's get to work!

References

Didier Dacunha-Castelle - Les mathématiques, le cerveau de l'âne et l'évolution - Cassini (2020)

Jean-Louis Krivine - Les décompilateurs (l'Univers en tête) - Calvage & Mounet (2024)

See also :

<https://www.irif.fr/~krivine>