Writing a Self-Reproducing Program

Amisi Kiarie

December 18, 2021

1 The Problem Defined

The problem given by Ken Thompson[1] is as follows: Write a source program that, when compiled and executed, will produce as output an exact copy of its source.

To understand the difficulty involved in the problem, let us try to specify it somewhat rigorously. Let $\mathscr{T}$ be the set of all strings and $\mathscr{U} \subset \mathscr{T}$ be the set of source programmes in our language, and $\mathscr{C}$ be the set of compiled programs. Then we may denote the *compilation* and *execution maps*

$$\mathbf{CMPL} : \mathscr{U} \to \mathscr{C} \text{ and } \mathbf{EXEC} : \mathscr{C} \to \mathscr{T}$$

respectively, provided we restrict $\mathscr{U}$ to programs that, when compiled and executed, only produce text output.

A *self-reproducing program* $S \in \mathscr{U}$ is one with the property

$$S = \mathbf{EXEC}(\mathbf{CMPL}(S)). \tag{1}$$

This may motivate the definition of the *interpretation map* $\mathbf{EVAL} : \mathscr{U} \to \mathscr{T}$ as

$$\mathbf{EVAL}(P) \coloneqq \mathbf{EXEC}(\mathbf{CMPL}(P)), \ P \in \mathscr{U}.$$

This allows us to rewrite (1) as

$$S = \mathbf{EVAL}(S).$$

In a word, S is an identity element of **EVAL**. By the same token, we have

$$\begin{aligned} S &= \mathbf{EVAL}(S) \\ &= \mathbf{EVAL}(\mathbf{EVAL}(S)) \\ &= \dots \\ &= \mathbf{EVAL}^N(S) \end{aligned}$$

for any positive integer N.

[1]*Reflections on Trusting Trust*, Turing Award Lecture, 1984.

2 Constructing S

2.1 A naive first attempt

Since we are focused on **EVAL** mainly, we may use the language python to arrive at a solution. Let P_0 be the one-line application

```
print("print()")
```

Then the output $\textbf{EVAL}(P_0)$ is

```
print()
```

If we attempt to modify P_0 so that the portion interior to the **PRINT** command[2] is included in the output, we obtain P_1

```
print("print(\"print()\")")
```

But this fails to solve our problem because $\textbf{EVAL}(P_1)$ is

```
print("print()")
```

However, we see that $\textbf{EVAL}(P_1) = P_0$, and it is not hard to see that we may easily construct the sequence $\{P_0, P_1, P_2, \dots\}$ with

$$\textbf{EVAL}(P_n) = P_{n-1}, \text{ for } n \geq 0.$$

Observe, then, that each step in writing a new 'upgrade' from $P_n \to P_{n+1}$ is something akin to applying $\textbf{EVAL}^{-1}$ to P_n. Now consider the following function:[3]

```
def upgrade(s):
    escaped = s.replace("\"", "\\\"")
    return f"print(\"{escaped}\")"
```

Note that if we write

```
upgrade("print()")
```

we obtain a string equivalent to P_0, i.e.

[2]The **PRINT** command maps strings from $\mathscr{T} \to \mathscr{T}$, and is defined in coordination with the representation map given below, so that $A = \textbf{PRINT}(\textbf{R}(A))$ for any string $A \in \mathscr{T}$.

[3]We are well aware that python offers both single- and double-quotes, but for the sake of illustrating our technique in a manner that is relatively language-agnostic, we have kept only to the latter.

```
print(upgrade("print()"))
```

outputs P_0. Moreover,

```
print(upgrade(upgrade("print()")))
```

outputs

```
print("print(\"print()\")")
```

which is simply P_1. The *upgrade* function thus maps $P_n \to P_{n+1}$.

Our aim is to obtain the program S such that $\textbf{EVAL}(S) = S$, and we have the following:

- $\textbf{EVAL}(P_n) = P_{n-1}$
- $\textit{upgrade}(P_n) = P_{n+1}$.

If we can somehow write a program that connects these two properties, we will obtain our self-reproducing S.

Reflection This attempt fails because it grows into an infinite loop of expansions. Perhaps we may learn something from $\{P_n\}$, other than humility! Our exercise with $\{P_n\}$ teaches us it is futile to use a literal string alone to construct a self-reproducing program because this string must contain itself *and* other things: if nothing else, it must contain the code that contains the **PRINT** command itself. A self-reproducing program S, therefore, cannot be constructed by printing a literal string only.

2.2 Self-recursive construction of S

Let us consider another form a self-reproducing program S may take. Suppose we have a block $\rho \in S$ which can be used *in some manner* to generate a literal representation of all of S except itself. We could combine ρ with another block $\varsigma \in S$ which can produce ρ and make use of ρ to produce itself.

More precisely, we can think of our program as having the form

$$S = \rho \cdot \varsigma, \tag{2}$$

where '$\cdot$' is the *concatenation operator*, and $\rho, \varsigma \in \mathscr{T}$:

- ρ is a block which can be used to produce $\varsigma = S - \rho$
- ς is a block which uses ρ to produce both ρ and itself, thereby producing S.

Thus ς sees itself through ρ and is a *mirror* or *self-reference* with respect to ς. This is what is meant by *self-recursive construction*.[4]

To make this more concrete, define $\mathbf{R} : \mathscr{T} \to \mathscr{T}$ to be the *representation map* so that for any $A \in \mathscr{T}$, we have

$$A = \mathbf{PRINT}(\mathbf{R}(A)).$$

We can now understand ρ as being the representation of ς,[5]

$$\rho = \mathbf{R}(\varsigma) \tag{3}$$

and then

$$\varsigma = \mathbf{PRINT}(\rho) \mathbin{.} \mathbf{PRINT}(\mathbf{R}^{-1}(\rho)). \tag{4}$$

(3) and (4), together with (2), give us a recursive specification for S. We may be assured it terminates because the representation map does not evaluate the expressions it is given, i.e.

$$\rho = \mathbf{R}\big(\mathbf{PRINT}(\rho) \mathbin{.} \mathbf{PRINT}(\mathbf{R}^{-1}(\rho))\big)$$

will simply be a string, the ρ's on the right will not be expanded further.

We now in a position to give this program in python. We begin by 'pretending' that we have a value for ρ, so we can construct ς on its basis. We must do this because according to (3), ρ is defined in terms of ς.

```
1  rho = b'' # TODO: fill in value
2
3  # sigma:
4  print(f"rho = {rho}", end="") # print(rho)
5  print(rho.decode("utf-8"))     # print(R^-1(rho))
```

Ignoring line 1, we may identify in lines 2–4 the components of (4) above:

- On lines 2–4 we have the equivalent of the command $\mathbf{PRINT}(\rho)$
- Line 5 is $\mathbf{PRINT}(\mathbf{R}^{-1}(\rho))$.

Execution of the program as it is simply produces

[4]The code which is offered in this document is based upon Thompson's program; but an example is available in the repository which I constructed before looking at his solution. However, it is messier than his construction so I chose to make use of his ideas in this article.

[5]The definitions given for ρ and ς are not meant to be exhaustive: the actual code may contain factors that are 'constant' with respect to these two, such as comments. So in effect we might have given a more full definition pair

$$\rho = C_0 \mathbin{.} \mathbf{R}(\varsigma) \mathbin{.} C_1$$

and

$$\varsigma = D_0 \mathbin{.} \mathbf{PRINT}(\rho) \mathbin{.} D_1 \mathbin{.} \mathbf{PRINT}(\mathbf{R}^{-1}(\rho)) \mathbin{.} D_2,$$

with constant factors $C_i, D_i \in \mathscr{T}$, but we felt that this would clutter the expressions.

```
rho = b''
```

Now we add the value for ρ. To obtain it we will take everything after the closing quote and encode it as a UTF-8 byte string. This is equivalent to applying the **R** in (3) above. Our final program is as follows:

```
rho = """

# sigma:
print(f"rho = {rho}", end="") # print(rho)
print(rho.decode('utf-8'))     # print(R^-1(rho))""".encode('utf-8')

# sigma:
print(f"rho = {rho}", end="") # print(rho)
print(rho.decode('utf-8'))     # print(R^-1(rho))
```

However, Thompson notes, purists will observe that this is not exactly a self-reproducing program: rather, it is the producer of one. Execution yields

```
rho = b'\n\n# sigma:\nprint(f"rho = {rho}", end="")...' # trimmed this line

# sigma:
print(f"rho = {rho}", end="") # print(rho)
print(rho.decode("utf-8"))     # print(R^-1(rho))
```

which is truly self-reproducing.