A different kind of fixed point theorem

MUSA Math Monday 3/17/2025

Patrick Lutz

Prologue: Fixed Point Theorems

Fixed point theorem: Every function $f : X \to X$ with property blah blah blah has a fixed point—i.e. there is some $x \in X$ such that $f(x) = x$

Example.
The Banach fixed point theorem (special case): Suppose $f : \mathbb{R} \to \mathbb{R}$ is a function such that for all $x, y \in \mathbb{R}$, $|f(x) - f(y)| < 0.9|x - y|$. Then f has a fixed point.

Proof. Pick any $a \in \mathbb{R}$ and define

$$x_0 = a, \quad x_1 = f(x_0), \quad x_2 = f(x_1), \quad \dots$$

Condition on f ensures:

- The sequence $x_0, x_1, x_2, \dots$ converges to some $x \in \mathbb{R}$
- For this x, $f(x) = x$ □

Other examples: Brouwer, Schauder, Kakutani, Caristi, Veblen, ...

Common feature: Proofs use limits/compactness/homotopy/etc.
i.e. topology

ixed point theorem: Every function $f : X \to X$ with property blah b
b as a fixed point—i.e. there is some $x \in X$ such that $f(x)$

Example

The Banach point theorem (special case): Suppo : $\mathbb{R} \to \mathbb{R}$ is a function such that ll $x, y \in \mathbb{R}$, $|f(x) - f(y)|$ $|x - y|$. Then f has a fixed point.

Proof. Pick any $a \in \mathbb{R}$ and de

$$x_0 = a, \quad x_1 = x_0), \quad f(x_1), \quad \ldots$$

Condition on f ensures

- The sequenc $x_1, x_2, \ldots$ converges to some $\mathbb{R}$
- For this $(x) = x$ □

Other ples: Brouwer, Schauder, Kakutani, Caristi, Veblen

C non feature: Proofs use limits/compactness/homotopy/etc.
i.e. topology

Today: A different kind of fixed point theorem with a very different proof

Chapter 1: Quines

Definition. A quine is a program that prints itself

Let's try to write a quine in Python!

Attempt 1: We need to print something. Here's how we do that in Python

Program: `print("Hello")`

Output: `Hello`

Attempt 2: Let's try to print something more relevant

Program: `print("print")`

Output: `print`

Attempt 3: Hmmm

Program: `print("print(\"print\")")`

Output: `print("print")`

Attempt 4: Hmmmmmmmmmmmmmmm...

Question: How can we possibly ever "catch up to ourselves"???

Question: How can we possibly ever "catch up to ourselves"???

A new idea: Consider the following "English language quine"

```
Write the following twice, the second time surrounded
by quotes: "Write the following twice, the second time
surrounded by quotes:"
```

Key idea: Self-reference

Actually, two key ideas:

(1) We can treat English language phrases as both instructions to follow and a sequence of words

(2) This allows us to apply an English language phrase to itself

Main theme of this talk: We will see these two ideas over and over again in different settings

Idea: Write a Python quine by trying to copy the English language quine

Program:

```
(lambda s: print(s + '(' + chr(34) + s + chr(34) + ')'))
("(lambda s: print(s + '(' + chr(34) + s + chr(34) + ')'))")
```

Question: What's going on here?

Answer. In Python:

(1) If `s` and `t` are strings, `s + t` is the concatenation of `s` and `t`

(2) `chr(34)` is another name for the character `"`

(3) `lambda x: f(x)` denotes the function $x \mapsto f(x)$

The first line of the program is a function that takes a string `s` and prints the string `s + (" + s + ")` i.e. the same string twice, but the second time in quotation marks (and parentheses)

The second line is the same as the first line, but in quotation marks (and parentheses) so it can be used as an input to the function on the first line

Idea: Write a Python quine by trying to copy the English language quine

Program:

```
(lambda s: print(s + ‘(’ + chr(34) + s + chr(34) + ‘)’))
("(lambda s: print(s + ‘(’ + chr(34) + s + chr(34) + ‘)’))")
```

The main theme, again:

Two key ideas: Programs can be viewed as both instructions to follow and sequences of characters.

i.e. code vs. data

This allows us to apply programs to themselves

There are many extensions and variations on quines

One interesting example: the qlock, invented by Martin Kleppe (aka @aemkei)

A program that prints itself and displays the current time by highlighting some of its characters

```
<script>

(r=()=>setInterval(t=>{for(j=o="\n",y=5;y--;document.body["inn"
+"erHTML"]="<pre>&lt"+(S="script>\n")+o+"\n\n&lt/"+S)for(x=-01;
63-!y>x++;o+=`(r=${r})()`[j++].fontcolor(c?"#FF0":"#444"))c=x/2
%4<3&&parseInt("odRFacb67o2vi5gmOZmwFNteohbOh3sw".slice(i="9"<(
D=Date()[16+(x/8|0)])?30:D*3,i+3),36)&1<<(x/2|0)%4+3*y},100))()

</script>
```

Chapter 2: A Fixed Point Theorem

Quines are cool, but...

Two flaws of our construction of a Python quine:

(1) It was ad-hoc and used tricks like `chr(34)` = `"`

(2) It doesn't obviously generalize to show how to build quines in other programming languages or build things like the qlock

Amazing fact: There is a general theorem that implies the existence of quines, qlocks, and much more in any reasonable programming language

The theorem: Kleene's fixed point theorem
(also known as the "second recursion theorem")

To state Kleene's fixed point theorem, we need a few definitions

First, fix a "reasonable" programming language

Assume: Each program P takes strings as input and outputs strings
We can view each program P as a string

An annoying detail: Not all computer programs ever stop and output an answer. Instead, some keep running forever.

Definition. A function $f : \text{Strings} \to \text{Strings}$ is computable if there is a program P such that for every string x, $P(x)$ eventually stops and outputs $f(x)$

Definition. Two programs P and Q are equivalent, written $P \approx Q$, if they have the same output behavior

i.e. for any input x, either $P(x)$ and $Q(x)$ both run forever, or they both eventually stop and output the same value

Example: The programs `P(x): output 5` and `Q(x): output 2 + 3` are equivalent

Theorem (Kleene's fixed point theorem): For any computable function f : Strings $\to$ Strings, there is some program P such that $f(P) \approx P$

This allows us to write programs that "know" their own code

General pattern:

(1) Let $f(x)$ be the function that, given a string x, outputs the program that you would write if you knew the program's code was x

(2) Kleene's fixed point theorem gives a program P such that $f(P) \approx P$

(3) So for any input x, $P(x)$ is equal to $f(P)(x)$—i.e. to the output of the program that you would write if you knew P was the code of the program!

Hard to understand abstractly. Let's look at some examples

Theorem (Kleene's fixed point theorem): For any computable function f : Strings $\to$ Strings, there is some program P such that $f(P) \approx P$

Application 1: Quines

Let's find a program P such that for any input x, $P(x)$ outputs P

Let f be the function which takes a string x and produces a program which ignores its input and always outputs x

For any "reasonable" programming language, this function will be computable

If P is a program such that P and $f(P)$ have the same behavior then since $f(P)$ always outputs P, so must P

Quibble: Before we said that a quine "prints" itself, but we just found a program that always "outputs" itself

Response to quibble: "Output" and "print" are not precise terms and roughly mean the same thing

Application 2: Qlocks

The notion of a qlock is not very mathematically precise, so here's a (slightly) more formal version:

Definition. A qlock is a program P such that for any $h < 24$, $m < 60$ and $s < 60$, $P(h, m, s)$ prints itself and displays `h : m : s` by changing the color of some of its characters

Let f be the function which takes a string x and produces a program which takes inputs h, m and s and prints x with `h : m : s` displayed by changing the color of some of x's characters

If P is a program such that P and $f(P)$ have the same behavior then P must be a qlock

Quibble: What if P is too short to display `h : m : s`?

Response to quibble: Modify f to make this impossible for any P which is a fixed point of f

Theorem (Kleene's fixed point theorem): For any computable function f : Strings $\to$ Strings, there is some program P such that $f(P) \approx P$

In general, this theorem allows us to write programs which know their own code

The theorem (and its variants) are extremely useful throughout several parts of logic. Especially proof theory, computability theory and descriptive set theory

One of those "magical" theorems
Like linearity of expectation, pigeonhole principle, compactness, etc

Recent example: MIP* = RE (which also resolved the Connes embedding conjecture)

Chapter 3: Russell's Paradox

Naive set theory:

(1) Everything is a set

(2) Sets contains elements, which are themselves sets

(3) Two sets are the same if and only if they have the same elements

(4) Any collection of sets that you can describe is also a set

Essentially invented by Frege, intended as a foundation for mathematics

An example of the last point: Given sets A and B, we can form the set

$$\{x \mid x \in A \text{ or } x \in B\}$$

also known as $A \cup B$

Another example: Given a set A, we can form the set

$$\{x \mid \text{for all } y \in x,\ y \in A\}$$

also known as $\mathcal{P}(A)$

Naive set theory:

(1) Everything is a set

(2) Sets contains elements, which are themselves sets

(3) Two sets are the same if and only if they have the same elements

(4) Any collection of sets that you can describe is also a set

Essentially invented by Frege, intended as a foundation for mathematics

Problem: It's inconsistent. Proved by Russell (sort of)

Proof. Let R be the set of sets that don't contain themselves. I.e.

$$R = \{x \mid x \notin x\}.$$

Question: does R contain itself?

If so: $R \in R \implies R \notin R$. Contradiction

If not: $R \notin R \implies R \in R$. Contradiction

A contradiction either way!

A different view of naive set theory: A set A can be thought of as a function Sets $\to \{0, 1\}$:

$$A(x) = \begin{cases} 0 & \text{if } x \notin A \\ 1 & \text{if } x \in A. \end{cases}$$

Translating Russell's paradox.

Russell set: $R = \{x \mid x \notin x\}$

Translation: R is the function defined by $R(x) = N(x(x))$ where $N\colon \{0, 1\} \to \{0, 1\}$ is the function defined by $N(0) = 1$ and $N(1) = 0$

Key question: Is $R \in R$?

Translation: What is the value of $R(R)$?

Key point: By definition, $R(R) = N(R(R))$.

I.e. $R(R)$ is a fixed point of the function N!

Main idea from previous slide:

(1) View sets as functions Sets $\to \{0, 1\}$

(2) Define a set R by $R(x) = N(x(x))$ where $N(0) = 1$ and $N(1) = 0$

(3) By definition, $R(R) = N(R(R))$. So $R(R)$ is a fixed point of N.

The main theme reappears.

Two key ideas: Sets can be viewed as both individual sets and as functions from sets to $\{0, 1\}$

This allows us to apply sets to themselves

Main idea from previous slide:

(1) View sets as functions Sets $\to \{0,1\}$

(2) Define a set R by $R(x) = N(x(x))$ where $N(0) = 1$ and $N(1) = 0$

(3) By definition, $R(R) = N(R(R))$. So $R(R)$ is a fixed point of N.

Another view of Russell's paradox: A machine for making fixed points

Key observation: There is nothing special about the function N in the argument above

More precisely: Given any function $F : \{0,1\} \to \{0,1\}$, we can define $R(x) = F(x(x))$.

Then by definition, $R(R) = F(R(R))$. In other words, $R(R)$ is a fixed point of F

Moreover, the same idea works to produce fixed points in many other settings

General framework: Suppose that the elements of the set X can be seen as functions $X \to Y$ and every such function is represented by some element of X

Then *every* function $F\colon Y \to Y$ has a fixed point

Proof: Given $x \in X$, let $\tilde{x}$ denote the function $X \to Y$ that it represents

Define a function $R\colon X \to Y$ by $R(x) = F(\tilde{x}(x))$

Let $r \in X$ be such that $\tilde{r} = R$

Note that $R(r) = F(\tilde{r}(r)) = F(R(r))$. Hence $R(r)$ is a fixed point of F

The theme again: Elements of X can be viewed as both elements of X and functions $X \to Y$ and this lets us apply elements of X to themselves

Since most sets Y have at least one function $F\colon Y \to Y$ without fixed points, the above argument is often useful in proofs by contradiction

I.e. make an assumption and show by the above argument that it leads to a fixed point which should exist

Another example: Cantor's diagonal argument

Theorem (Cantor). There is no surjection $g\colon \mathbb{N} \to \mathbb{R}$

Proof. Suppose there was. Then we can think of each number $n \in \mathbb{N}$ as corresponding to a function $\tilde{n}\colon \mathbb{N} \to \{0, 1, \ldots, 9\}$ by

$$\tilde{n}(k) = k^{\text{th}} \text{ digit after the decimal point of } g(n)$$

Since g is a surjection, every function $\mathbb{N} \to \{0, 1, \ldots, 9\}$ is represented in this way (ignoring a few exceptions...).

By the proof we just saw, this gives a contradiction as long as there is some function $\{0, 1, \ldots, 9\} \to \{0, 1, \ldots, 9\}$ with no fixed points

More concretely: define a function $R\colon \mathbb{N} \to \{0, 1, \ldots, 9\}$ by

$$R(n) = \tilde{n}(n) + 1 \mod 10$$

If $r \in \mathbb{N}$ is such that $\tilde{r} = R$ then

$$\tilde{r}(r) = R(r) = \tilde{r}(r) + 1 \mod 10$$

which is impossible

Chapter 4: Proving Kleene's Theorem

Theorem (Kleene's fixed point theorem): For any computable function f : Strings $\to$ Strings, there is some program P such that $f(P) \approx P$

Some assumptions about the programming language:

Assumption 1: Each program is a string

Assumption 2: Each program P takes one or more strings as input and outputs a string

Assumption 3: There is a program `Run(x, y)` such that for any program P and string x, `Run(P, x)` is equal to the output of $P(x)$

If $P(x)$ runs forever, so does `Run(P, x)`

If P takes a number of inputs other than 1, then `Run(P, x)` can have any behavior

Comment: `Run` is essentially an interpreter for the programming language

Theorem (Kleene's fixed point theorem): For any computable function f : Strings $\to$ Strings, there is some program P such that $f(P) \approx P$

Proof. Let R be the following program:

```
R(x):
  Output the following program:
    1. Given input y
    2. Let a = Run(x, x)
    3. Output Run(f(a), y)
```

Intuitively: $R(x)$ outputs a program P such that $P(y) = f(x(x))(y)$
I.e. $R(x) \approx f(x(x))$ (assuming x represents a program and $x(x)$ doesn't run forever)

Important point: $R(x)$ always eventually stops and outputs something
I.e. $R(x)$ doesn't run forever

Now let $P = R(R)$. As above, $R(R) \approx f(R(R))$, so $P \approx f(P)$ $\square$

Mysterious!

Theorem (Kleene's fixed point theorem): For any computable function f : Strings $\rightarrow$ Strings, there is some program P such that $f(P) \approx P$

(Obvious?) Question: Why doesn't this theorem lead to a contradiction? I.e. use a computable function f such that for all programs P, $f(P) \not\approx P$

Answer: It is hard to build such a function f because it is hard to determine the behavior of a program

The main theme, one last time:

Programs can be viewed as both functions on strings and strings

This allows us to apply programs to themselves