A REVIEW OF THE GÖDEL FIXED-POINT THEOREM WITH GENERALIZATIONS AND APPLICATIONS

JOEL DAVID HAMKINS

Abstract. This is a brief overview of the Gödel fixed-point lemma, along with several generalizations and applications, written for use in a lecture in the graduate Philosophy of Logic seminar at Oxford in Hilary term 2021. Parts of this text are adapted from [Ham21].

1. Introduction

Let us work in the usual language of arithmetic $\{+, \cdot, 0, 1, <\}$, and I shall assume that we have already well undertaken the arithmetization process, by which we interpret various finite combinatorial objects and constructions within arithmetic. Essentially any finite combinatorial concept, whether it concerns strings of symbols or finite graphs or Turing machine computations, can be encoded within numbers in such a way that the finite combinatorial manipulations of those objects and concepts can be simulated within arithmetic, just as your Word document is encoded by a long binary sequence of 0s and 1s inside your computer, with all your edits and copy/past manipulations performed ultimately by the arithmetic of logic gates. In particular, we assume that for every finite combinatorial object s, we have fixed an interpretation of it in numbers, associating with s a number $\ulcorner s \urcorner$, the Gödel code of s.

2. Gödel's fixed-point lemma

The fixed-point lemma is an enigma—a mathematical mystery, a logical labyrinth that shows how self-reference, the stuff of nonsense and confusion, sneaks explicitly into our beautiful number theory. It continually amazes me.

Lemma 1 (The fixed-point lemma)**.** *For any formula $\varphi(x)$ with one free variable, there is a sentence ψ such that*

$$\mathrm{PA} \vdash \psi \leftrightarrow \varphi\big(\ulcorner \psi \urcorner\big).$$

The sentence ψ asserts that "φ holds of my Gödel code."

There is a certain irritating matter of notation, for when we write $\varphi\big(\ulcorner \psi \urcorner\big)$ what we should actually mean is $\varphi(\underline{k})$, where $k = \ulcorner \psi \urcorner$ is the Gödel code of ψ and $\underline{k}$ is the term $1 + \cdots + 1$, with k many 1s. The issue is that $\ulcorner \psi \urcorner$ is a number rather than a syntactic expression, but in order for it to be sensible to prove the equivalence, we need the substitution instance to be a sentence in the language. We are not merely claiming that the equivalence is true in the standard model, but provable in PA. So kindly allow this small notational infelicity, with the understanding that whenever we substitute a number into a formula $\varphi(k)$, that we should do so using $\varphi(\underline{k})$ with the term $\underline{k} = 1 + \cdots + 1$. Nevertheless, I shall try to use the underbar numeral notation whenever doing so does not unduly encumber the notation.

Proof. Let sub be the substitution function, defined such that

$$\mathrm{sub}\big(\ulcorner\varphi(x)\urcorner, m\big) = \ulcorner\varphi(\underline{m})\urcorner,$$

where $\varphi(x)$ is any formula with one free variable x, for which we have substituted the numeral $\underline{m}$. The sub function is a primitive recursive function, and it is representable in the language of arithmetic in accordance with the principle of arithmetization. Consider now any formula $\varphi(x)$ with one free variable x. Let $\theta(x) = \varphi\big(\mathrm{sub}(x,x)\big)$, and let $n = \ulcorner\theta(x)\urcorner$. Finally, let $\psi = \theta(\underline{n})$, which is a sentence in the language of arithmetic. Putting all this together, we observe in PA the following equivalences:

$$\begin{aligned}
\psi &\leftrightarrow \theta(\underline{n}) \\
&\leftrightarrow \varphi\big(\mathrm{sub}(\underline{n},\underline{n})\big) \\
&\leftrightarrow \varphi\big(\mathrm{sub}(\ulcorner\theta(x)\urcorner,\underline{n})\big) \\
&\leftrightarrow \varphi\big(\ulcorner\theta(\underline{n})\urcorner\big) \\
&\leftrightarrow \varphi\big(\ulcorner\psi\urcorner\big).
\end{aligned}$$

Thus, the sentence ψ has the desired fixed-point property. □

With the fixed-point lemma, we can readily construct sentences that seem to refer to themselves. For example, if $P(x)$ asserts that x is the Gödel code of an axiom of PA, then a fixed point ϕ, where $\phi \leftrightarrow P(\ulcorner\phi\urcorner)$, can be read as the assertion, "This sentence is an axiom of PA." Is it true? No, since PA has no axioms of exactly that form. If $S(x)$ asserts that x is the Gödel code of a sentence, then a fixed point σ, where $\sigma \leftrightarrow S(\ulcorner\sigma\urcorner)$, can be read as the assertion, "This is a sentence." And indeed it is.

An application to the Gödel incompleteness theorem. Similar ideas lie at the heart of the fixed-point proof of the incompleteness theorem. Suppose that T is a theory in the language of arithmetic, containing and possibly extending PA, and that the axioms of T can be enumerated by a computable procedure. It follows by the arithmetization of syntax that the predicate $\mathrm{Pr}_T(x)$, asserting that x is the Gödel code of a sentence that is provable from T, is expressible in the language of arithmetic. By the fixed-point lemma, therefore, there is a sentence ψ, known as the *Gödel sentence*, such that

$$\mathrm{PA} \vdash \psi \leftrightarrow \neg\mathrm{Pr}_T(\ulcorner\psi\urcorner).$$

If you reflect upon this situation carefully, you will come to the amazing conclusion that this sentence ψ asserts exactly its own unprovability. The Gödel sentence ψ asserts, "This sentence is not provable in T."

Perhaps you have heard of the Liar paradox, the sentence "This sentence is false," which asserts its own falsity. The Liar sentence, it seems, cannot be true, for then it would also be false; and it cannot be false, for then it would also be true. So which is it? It is the Liar paradox. The Gödel sentence ψ is not the Liar, but a cousin of it, using provability in place of truth—or, more accurately, using unprovability in place of falsity. This is a difference that matters, since although the Liar sentence can be formulated easily in natural language, the semantic notions of truth and falsity are not subject to arithmetization, and we seem unable meaningfully to express the Liar sentence in the language of arithmetic. Indeed, it must be impossible to

do so, since the paradox would turn into outright contradiction, as all arithmetic assertions have truth values.

The Gödel sentence, in contrast, is expressible in arithmetic because it replaces the semantic truth concept with its syntactic analogue, provability. Since provability is amenable to arithmetization, Gödel is able to construct his sentence, "This sentence is not provable in T." This is a sentence in the formal language of arithmetic, a mathematical statement like any other in the realm of arithmetic. Thus, Gödel taps the strange logic of the Liar. Whereas the Liar sentence leads to a paradox or outright contradiction, the Gödel sentence leads instead exactly to the conclusions of the incompleteness theorem.

Namely, we prove the incompleteness theorem as follows: Assume that T is a consistent, computably axiomatizable theory of arithmetic extending PA. If the Gödel sentence ψ were provable in T, then by inspecting the proof, we could also prove that ψ was provable. That is, T would prove $\mathrm{Pr}_T(\ulcorner\psi\urcorner)$. But this is provably equivalent to $\neg\psi$, and so T will have proved both ψ and $\neg\psi$, thereby revealing inconsistency, contrary to assumption. So T does not prove ψ. But this is precisely what ψ itself asserts. So ψ is a true sentence that is not provable in T. We have therefore established the following version of the first incompleteness theorem.

Theorem 2 (First incompleteness theorem, Gödel, 1931)**.** *Every consistent, computably axiomatizable theory of arithmetic admits true but unprovable statements.*

3. Finite self-referential schemes

Let us now consider a generalization of the fixed-point lemma to the case of finite systems of self-referential assertions. We begin with the two-dimensional case, due to Montague [Mon62]; see also [Lin03, chapter 1], as well as various works of Raymond Smullyan, who is to be credited with popularizing many of these fixed-point arguments.

Lemma 3 (Double Fixed Point Lemma)**.** *Suppose that $A(x,y)$ and $B(x,y)$ are two formulas in the language of arithmetic, then there are sentences ϕ and ψ such that* PA *proves the equivalences*

$$\phi \leftrightarrow A(\ulcorner\phi\urcorner, \ulcorner\psi\urcorner)$$

and

$$\psi \leftrightarrow B(\ulcorner\phi\urcorner, \ulcorner\psi\urcorner).$$

Proof. Let Sub be the binary substitution operator, the primitive recursive function such that $\mathrm{Sub}(\ulcorner\eta(x,y)\urcorner, n, m) = \ulcorner\eta(\underline{n}, \underline{m})\urcorner$. Let $\theta_1(x,y) = A(\mathrm{Sub}(x,x,y), \mathrm{Sub}(y,x,y))$ and $\theta_2(x,y) = B(\mathrm{Sub}(x,x,y), \mathrm{Sub}(y,x,y))$. Let $n = \ulcorner\theta_1(x,y)\urcorner$ and $m = \ulcorner\theta_2(x,y)\urcorner$. Finally, let $\phi = \theta_1(\underline{n}, \underline{m})$ and $\psi = \theta_2(\underline{n}, \underline{m})$.

Observe that

$$\begin{aligned}
\phi &\leftrightarrow \theta_1(\underline{n}, \underline{m}) \\
&\leftrightarrow A(\mathrm{Sub}(\underline{n}, \underline{n}, \underline{m}), \mathrm{Sub}(\underline{m}, \underline{n}, \underline{m})) \\
&\leftrightarrow A(\ulcorner\theta_1(\underline{n}, \underline{m})\urcorner, \ulcorner\theta_2(\underline{n}, \underline{m})\urcorner) \\
&\leftrightarrow A(\ulcorner\phi\urcorner, \ulcorner\psi\urcorner).
\end{aligned}$$

Also observe

$$\begin{aligned}\psi &\leftrightarrow \theta_2(\underline{n},\underline{m})\\ &\leftrightarrow B(\mathrm{Sub}(\underline{n},\underline{n},\underline{m}),\mathrm{Sub}(\underline{m},\underline{n},\underline{m}))\\ &\leftrightarrow B(\ulcorner\theta_1(\underline{n},\underline{m})\urcorner,\ulcorner\theta_2(\underline{n},\underline{m})\urcorner)\\ &\leftrightarrow B(\ulcorner\phi\urcorner,\ulcorner\psi\urcorner),\end{aligned}$$

as desired. □

Note that we can arrange that ϕ and ψ are distinct simply by ensuring that $\theta_1(n,m)$ and $\theta_2(n,m)$ are not syntactically the same sentence, such as by replacing $\theta_1(x,y)$ with its conjunction, if necessary, while ensuring that $\theta_2(x,y)$ does not have such a form.

The lemma easily generalizes to any size system and indeed, to infinite systems of fixed points.

Exercise 3.1. *Formulate and prove the triple fixed-point lemma and more generally, the n-fold fixed point lemma.*

An application to nonindependent disjunctions of independent sentences. As an application, let us consider the following question, posed on MathOverflow by Alex Gavrilov [Gav11]:

Question 4. *If ϕ and ψ are Π^0_1 sentences independent of* PA*, must $\phi\vee\psi$ also be independent of* PA*?*

My answer, posted at [Ham11], was: not necessarily. Specifically, we can make a counterexample using the double fixed-point lemma. Using that lemma, we may produce two distinct sentences ϕ and ψ such that

> ϕ asserts that for every proof of ϕ, there is a smaller proof of ψ, and
> ψ asserts that for every proof of ψ, there is a smaller proof of ϕ.

Each of these statements has complexity Π^0_1. Let me argue that they are independent. Observe first that both ϕ and ψ are true in the standard model $\mathbb{N}$. If ϕ were false, then there would be a standard proof of ϕ, having no smaller standard proof of ψ. In particular, ϕ would be a provable, false statement, contradicting $\mathbb{N}\models$ PA. An essentially similar argument applies to ψ. Second, observe that neither is provable in PA. If ϕ were provable, then there would be a standard proof of ϕ, and thus there would have to be a smaller standard proof of ψ, and so ψ would be true, and so there would be an even smaller standard proof of ϕ. Thus, there could be no smallest proof of ϕ, a contradiction. And the same for ψ. So these are both true and unprovable sentences, hence independent.

Finally, observe that the disjunction $\phi\vee\psi$ is provable. If both ϕ and ψ fail in a model of PA, then that model would have proofs of both ϕ and ψ, but neither statement could have the smallest proof, for if it did, then the other statement would be true, contrary to assumption. This contradicts PA, since the smallest proof of one of them must be smaller than any proof of the other.

In summary, we have constructed Π^0_1 sentences ϕ and ψ that are independent of PA, such that $\phi\vee\psi$ is provable in PA. □

4. Gödel-Carnap fixed point lemma

Let us now extend the Gödel fixed-point lemma to the Gödel-Carnap fixed-point lemma, which treats formulas rather than merely sentences. This form of the fixed-point lemma is no more difficult to prove than the original, and it subsumes the double fixed-point and finite system fixed-point generalizations.

Lemma 5. *For any formula $\varphi(x, y)$, there is a formula $\psi(x)$ such that*

$$\text{PA} \vdash \forall x \left[\psi(x) \leftrightarrow \varphi\big(x, \ulcorner\psi\urcorner\big)\right].$$

Proof. The proof is nearly identical to the Gödel fixed-point lemma, since it does not matter much that φ has other free variables. One needs a slightly more refined version of the substitution function, to indicate which variable it is that one is substituting. □

Of course, we can just as easily handle more than one extra free variable.

Lemma 6. *For any formula $\varphi(x_0, \ldots, x_k, y)$, there is a formula $\psi(x_0, \ldots, x_k)$ such that*

$$\text{PA} \vdash \forall x_0, \ldots, x_k \left[\psi(x_0, \ldots, x_k) \leftrightarrow \varphi\big(x_0, \ldots, x_k, \ulcorner\psi\urcorner\big)\right].$$

Deriving the double fixed-point lemma as a consequence. We can easily prove the double fixed-point lemma as a consequence of the Gödel-Carnap fixed-point lemma. Namely, given formulas $A(u, v)$ and $B(u, v)$, simply find a formula $\psi(y)$ such that $\psi(0) \leftrightarrow A(\ulcorner\psi(0)\urcorner, \ulcorner\psi(1)\urcorner)$ and $\psi(1) \leftrightarrow B(\ulcorner\psi(0)\urcorner, \ulcorner\psi(1)\urcorner)$. Note that any assertions about $\ulcorner\psi(0)\urcorner$ and $\ulcorner\psi(1)\urcorner$ can be seen as an assertion about $\ulcorner\psi\urcorner$. So we can take $\psi(0)$ and $\psi(1)$ as the solution to the double fixed-point system.

Similarly, we may naturally view the Gödel-Carnap fixed-point lemma as providing a generalizaton of the double fixed-point, the triple fixed-point and indeed the n-fold fixed point lemma to an infinitary version, indexed by the parameter x.

An application to the provability version of Yablo's paradox. Yablo's paradox involves an infinite list of statements θ_0, θ_1, θ_2, and so on, where each θ_n asserts the falsity of all the later sentences θ_k for $k > n$. This situation is paradoxical, since if any given θ_n were true, then all the later θ_k would have to be false, but in this case, θ_{n+1} would have also to be true, contrary to our claim that it is false; so none of the sentences can be true, but in this case, any given one of them would have to be true. Crazytown!

The paradox is proposed as a version of the Liar paradox omitting self-reference. To my way of thinking, however, this version of the Liar paradox does involve self-reference, since the sentences are referring to the enumeration of sentences, of which they are a part. Really we should view these sentences as arising as instantiations $\theta(n)$ of a single formula $\theta(x)$, and the Yablo supposition is that

$$\theta(n) \leftrightarrow \forall k > n \; \neg T(\theta(k)),$$

which makes reference to the formula θ. So I view this paradox as still involving self-reference.

What I would like to do here is to consider the provability analogue of Yablo's paradox, obtained via arithmetization by the Gödel-Carnap fixed point lemma. Namely, by the fixed-point lemma, there is a formula $\theta(x)$, where each instance asserts the nonprovability of all the later instances, like this:

$$\text{PA} \vdash \forall x \;\; \left[\theta(x) \leftrightarrow \forall k > x \; \neg \Pr{}_{\text{PA}}\big(\ulcorner\theta(\underline{k})\urcorner\big)\right].$$

Let us analyze these sentences.

Theorem 7. *The Yablo provability assertions $\theta(n)$, defined above, are all provably equivalent to one another and provably equivalent, individually, to their own nonprovability and also, individually, to* Con(PA).

Proof. Because each $\theta(x)$ asserts the nonprovability of all $\theta(k)$ for $k > n$, the scope of this quantifier is reduced with larger n, and so every instance $\theta(x)$ implies all the later instances $\theta(k)$ for $k > x$, and provably so. In other words, PA can prove that once one of them is true, then all the later ones are also true. From this it follows that provably, if any $\theta(n)$ is provable, then also all the later $\theta(k)$ are provable, and consequently, in light of what $\theta(n)$ asserts, also that $\neg\theta(n)$ is provable. Thus, PA can prove that if any $\theta(n)$ is provable, then so is the negation $\neg\theta(n)$, and consequently $\neg\,\mathrm{Con(PA)}$. Therefore, Con(PA) implies that none of them is provable, and consequently, in light of what they assert, also that all of them are true. Conversely, $\neg\,\mathrm{Con(PA)}$ implies that they are all provable and consequently, again in light of what they assert, also that they are all false.

So the situation is that in any model of PA, either all of the Yablo provability assertions are true and unprovable, or all of them are false and provable, depending respectively on whether Con(PA) holds or not. In particular, each of them is equivalent to the others and to Con(PA) and to their own nonprovability. □

In this sense, we can view each Yablo provability assertion $\theta(n)$ as a version of the Gödel sentence, because $\theta(n)$ is provably equivalent to its own nonprovability.

5. Kleene recursion theorem

Let us now consider the computability version of the fixed-point idea. Let us assume that every number e can be interpreted as a Turing machine program. Let φ_e be the function computed by program e. We can interpret any program e as computing a function of any desired arity.

Theorem 8 (Kleene recursion theorem)**.** *For any computable function f, there is a Turing machine program e such that e and $f(e)$ compute the same function.*

Proof. If we have a program p computing a binary function $\varphi_p(u, x)$, then for any particular u, we can produce a program $s(p, u)$ that computes the slice of the function where u has been fixed, so that $\varphi_{s(p,u)}(x) = \varphi_p(u, x)$. This is the computability analogue of the substitution operation, and there is a primitive recursive such function s.

For any computable function f, let $g(u, x) = \varphi_{f(s(u,u))}(x)$, which is a computable function computed by some program d, so that $g(u, x) = \varphi_d(u, x)$. Let $e = s(d, d)$. Now simply observe that $\varphi_e(x) = \varphi_{s(d,d)}(x) = \varphi_d(d, x) = g(d, x) = \varphi_{f(s(d,d))}(x) = \varphi_{f(e)}(x)$. So the programs e and $f(e)$ compute the same function, as desired. □

The Kleene recursion theorem is widely used in computability theory arguments, particularly those involved in investigating the nature of the Turing degrees. Just as the Gödel fixed-point lemma enables one to find sentences that refer to themselves, the Kleene recursion theorem allows one to construct programs that refer to themselves and in particular, make computational decisions based on their own behavior on other input.

An application involving computable numbers. Let me illustrate a use of the recursion theorem in application to the notion of computable real numbers, introduced by Turing [Tur36]. (This presentation is adapted from chapter 6 of [Ham21]; one might also see the further discussion on my blog post at [Ham18].)

Alan Turing essentially founded the subject of computability theory in his classic 1936 paper, "On Computable Numbers, with an Application to the Entscheidungsproblem," in which he achieves so much: He defines and explains his machines; he describes a universal Turing machine; he shows that one cannot computably determine the validities of any sufficiently powerful formal proof system; he shows that the halting problem is not computably decidable; he argues that his machine concept captures our intuitive notion of computability; and he develops the theory of computable real numbers.

That last concern was the title focus of the article, and Turing defines, in the very first sentence, that a computable real number is one whose decimal expansion can be enumerated by a finite procedure, by what we now call a Turing machine, and he elaborates on and confirms this definition later in detail. He proceeds to develop the theory of computable functions of computable real numbers. In this theory, one does not apply the functions directly to the computable real numbers themselves, but rather to the programs that compute those numbers. In this sense, a computable real number is not actually a kind of real number. Rather, to have a computable real number in Turing's theory is to have a program—a program for enumerating the digits of a real number.

I should like to criticize Turing's approach, which is not how researchers today define the computable numbers. Indeed, Turing's approach is now usually described as one of the natural but ultimately mistaken ways to proceed with this concept. One main problem with Turing's account, for example, is that with this formulation of the concept of computable numbers, the operations of addition and multiplication on computable real numbers turn out not to be computable. Let me explain. The basic mathematical fact in play is that the digits of a sum of real numbers $a+b$ are not continuous in the digits of a and b separately; one cannot necessarily say with certainty the initial digits of $a+b$, knowing only finitely many digits, as many as desired, of a and b.

To see this, consider the following sum $a+b$:

$$\begin{array}{rr} & 0.343434343434\ldots \\ + & 0.656565656565\ldots \\ \hline & 0.999999999999\ldots \end{array}$$

The sum has 9 in every place, which is fine, and we could accept either $0.999\ldots$ or $1.000\ldots$ as correct, since either is a decimal representation of the number 1. The problem, I claim, is that we cannot know whether to begin with 0.999 or 1.000 on the basis of only finitely many digits of a and b. If we assert that $a+b$ begins with 0.999, knowing only finitely many digits of a and b, then perhaps the later digits are all 7s, which would force a carry term, causing all those 9s to roll over to 0, with a leading 1. Thus, our answer would have been wrong. If, alternatively, we assert that $a+b$ begins with 1.000, knowing only finitely many digits of a and b, then perhaps the later digits of a and b are all 2s, which would mean that $a+b$ is definitely less than 1. Thus, in any case, no finitely many digits of a and b can

justify an answer for the initial digits of $a+b$. Therefore, there is no algorithm to compute the digits of $a+b$ continuously from the digits of a and b separately.

The claim that there can be no computable algorithm for computing the digits of $a+b$, given the programs that compute a and b separately, however, is a somewhat more subtle claim, and this is where this example involves the Kleene recursion theorem. Namely, let $a = .343434\ldots$, and then consider a program to enumerate a number b, which begins with 0.656565 and keeps repeating 65 until the addition program has given the initial digits for $a+b$, at which point our program for b switches either to all 7s or all 2s, in such a way so as to refute the result. The Kleene recursion theorem is used in order to construct such a self-referential program for b, since the number b that we are ultimately enumerating depends on the what the supposed addition program gives us as the initial digits of $a+b$. Specifically, for any program d, let $f(d)$ be the program that enumerates the digits $0.656565\ldots$ while also inspecting the digits of $a+x$ where x is the real enumerated by d, whatever it is, and then switches either to all 7s or all 2s depending on whether $a+x$ begins with $1.0\ldots$ or larger or whether it begins $0.9\ldots$ or less, respectively. If e is a fixed-point provided by the Kleene theorem, so that e and $f(e)$ enumerate the same digits, then the number b enumerated by e has exactly the features requested.

An application involving the universal algorithm. Let me describe a second application of the Kleene recursion theorem with an elementary version of the universal algorithm.

Theorem 9. *For any consistent theory T extending* PA*, there is a Turing machine program e, which we can write down, such that for any partial function $f : \mathbb{N} \to \mathbb{N}$, there is a model of the theory, such that if we run the program e on input n inside the model, then for $n \in \operatorname{dom}(f)$ the result is $f(n)$, but if $n \notin \operatorname{dom}(f)$, then the program does not halt.*

Proof. The program e searches for a proof from T of a statement of the form "the function computed by e is not precisely the function determined entirely by these specific input/output pairs: $(k_0, n_0), \ldots, (k_r, n_r)$." If such a proof is found, then the program proceeds to halt on exactly those inputs k_i giving output n_i; for inputs not of the form k_i on that list, the program proceeds to fill the output tape with 1s. We use the Kleene recursion theorem in order to know that indeed there such a program e defined by this self-referential recursion.

In the standard model, I claim, the program will never halt. If it ever did halt, then it will have done so because it found such a proof, but then proceeded to halt anyway on exactly those inputs with exactly those outputs. By inspecting the computation, we would be able to prove this is the behavior, and this would show that T is inconsistent. So in the standard model, there are no such proofs to be found.

But precisely because of this, it follows that for any particular desired finite list of input/output behavior $(k_0, n_0), \ldots, (k_r, n_r)$, it is consistent with T that the program e halts on these k_i with output n_i, and diverges on all other input. Therefore, for any partial function $f : \mathbb{N} \to \mathbb{N}$, it is finitely consistent with T that the operation of e is in accordance with f. And so the whole theory is consistent. So there is a model $M \models T$ in which the function computed by e is altogether in accordance with f on standard input n. □

An application to Quine programs and Ouroborous chains. Finally, let me mention the curious phenomenon of the *Quine* programs, which are programs that give as output their own source code. Imagine a Turing machine program that prints out its own source code on the tape, or a Java program that prints itself out on the screen. Since this output can be used in further computation, a Quine is a program able to handle itself computationally.

Programmers often compete to produce very short Quines. Here are some examples, adapted from [Wik18]. This Python code gives itself as output:

```
s = 's = %r; print(s%%s)'; print(s%s)
```

This C program gives itself as output:

```
int main(){char*s="int main(){char*s=%c%s%c;printf(s,34,s,34);return 0;}";
printf(s,34,s,34);return 0;}
```

And here is an SQL Quine:

```
SELECT REPLACE(REPLACE('SELECT REPLACE(REPLACE("$",CHAR(34),CHAR(39)),CHAR(36),"$")
AS Quine',CHAR(34),CHAR(39)),CHAR(36),'SELECT REPLACE(REPLACE("$",CHAR(34),CHAR(39)),CHAR(36),"$")
AS Quine') AS Quine
```

I should like to point out that the Quine concept is not actually computability theoretic, in the sense that it does not respect the equivalence of programs, and it is sensitive to the computational model. If you replace a program with another that computes the same function, or with an equivalent program in a different computational language, then it may no longer be a Quine.

Nevertheless, Quines are fascinating. Let us prove, using the Kleene recursion theorem, that they exist for any sufficiently powerful programming language.

Theorem 10. *Every Turing-complete programming language admits a Quine, a program e that gives itself as output.*

Proof. Let $f(e)$ be a program that gives e as output. By the recursion theorem, there is a program e such that e and $f(e)$ give the same output. So program e gives e as output. □

Since the proof of the recursion theorem is constructive, this argument can be used to produce explicit Quines. And indeed, one can observe in the Quine examples we gave earlier the $\mathrm{sub}(x, x)$ feature that is core to the fixed-point lemmas and the recursion theorem.

The TEX typesetting language has programming features that make it Turing complete, and so actually there are Quines in TEX. The following example is due to Péter Szabó. Here is the TEX source code:

```
\def\T{
\tt \hsize 32.5em\parindent 0pt\def \S {\def \S ##1>{}}\S \string
\def \string \T \string {\par \expandafter \S \meaning \T \string
}\par \expandafter \S \meaning \T \footline {} \end }
\tt \hsize 32.5em\parindent 0pt\def \S {\def \S ##1>{}}\S \string
\def \string \T \string {\par \expandafter \S \meaning \T \string
}\par \expandafter \S \meaning \T \footline {} \end
```

And this is the output when that document is processed by TEX:

```
\def\T{
\tt \hsize 32.5em\parindent 0pt\def \S {\def \S ##1>{}}\S \string
\def \string \T \string {\par \expandafter \S \meaning \T \string
}\par \expandafter \S \meaning \T \footline {} \end }
\tt \hsize 32.5em\parindent 0pt\def \S {\def \S ##1>{}}\S \string
\def \string \T \string {\par \expandafter \S \meaning \T \string
}\par \expandafter \S \meaning \T \footline {} \end
```

It is possible to construct more elaborate kinds of fixed points. An *Ouroboros* program, or Quine cycle, is a program in one language, that produces output which is a program in another language, whose output is the original program (or perhaps it proceeds for several steps in several additional languages). For example, below at left (from [Wik18]) is some Java source code, which has as output the C++ code at right, which has as output the original Java code.

```
public class Quine
{
  public static void main(String[] args)
  {
    char q = 34;
    String[] l = {
    "    ",
    "============<<<<<<<< C++ Code >>>>>>>>============",
    "#include <iostream>",
    "#include <string>",
    "using namespace std;",
    "",
    "int main(int argc, char* argv[])",
    "{",
    "    char q = 34;",
    "    string l[] = {",
    "    };",
    "    for(int i = 20; i <= 25; i++)",
    "        cout << l[i] << endl;",
    "    for(int i = 0; i <= 34; i++)",
    "        cout << l[0] + q + l[i] + q + ',' << endl;",
    "    for(int i = 26; i <= 34; i++)",
    "        cout << l[i] << endl;",
    "    return 0;",
    "}",
    "============<<<<<<<< Java Code >>>>>>>>==========",
    "public class Quine",
    "{",
    "  public static void main( String[] args )",
    "  {",
    "    char q = 34;",
    "    String[] l = {",
    "    };",
    "    for(int i = 2; i <= 9; i++)",
    "        System.out.println(l[i]);",
    "    for(int i = 0; i < l.length; i++)",
    "        System.out.println( l[0] + q + l[i] + q + ',' );",
    "    for(int i = 10; i <= 18; i++))",
    "        System.out.println(l[i]);",
    "  }",
    "}",
    };
    for(int i = 2; i <= 9; i++)
        System.out.println(l[i]);
    for(int i = 0; i < l.length; i++)
        System.out.println( l[0] + q + l[i] + q + ',' );
    for(int i = 10; i <= 18; i++)
        System.out.println(l[i]);

  }
}
```

```
#include <iostream>
#include <string>
using namespace std;

int main(int argc, char* argv[])
{
    char q = 34;
    string l[] = {
    "    ",
    "============<<<<<<<< C++ Code >>>>>>>>============",
    "#include <iostream>",
    "#include <string>",
    "using namespace std;",
    "",
    "int main(int argc, char* argv[])",
    "{",
    "    char q = 34;",
    "    string l[] = {",
    "    };",
    "    for(int i = 20; i <= 25; i++)",
    "        cout << l[i] << endl;",
    "    for(int i = 0; i <= 34; i++)",
    "        cout << l[0] + q + l[i] + q + ',' << endl;",
    "    for(int i = 26; i <= 34; i++)",
    "        cout << l[i] << endl;",
    "    return 0;",
    "}",
    "============<<<<<<<< Java Code >>>>>>>>==========",
    "public class Quine",
    "{",
    "  public static void main(String[] args)",
    "  {",
    "    char q = 34;",
    "    String[] l = {",
    "    };",
    "    for(int i = 2; i <= 9; i++)",
    "        System.out.println( l[i] );",
    "    for(int i = 0; i < l.length; i++)",
    "        System.out.println(l[0] + q + l[i] + q + ',');",
    "    for(int i = 10; i <= 18; i++)",
    "        System.out.println(l[i]);",
    "  }",
    "}",
    };
    for(int i = 20; i <= 25; i++)
        cout << l[i] << endl;
    for(int i = 0; i <= 34; i++)
        cout << l[0] + q + l[i] + q + ',' << endl;
    for(int i = 26; i <= 34; i++)
        cout << l[i] << endl;
    return 0;
}
```

Diverse longer Ouroboros chains have been constructed.

References

[Gav11] Alex Gavrilov. *The disjunction property in Peano Arithmetic?* MathOverflow question. 2011. https://mathoverflow.net/q/63160.

[Ham11] Joel David Hamkins. *The disjunction property in Peano Arithmetic?* MathOverflow answer. 2011. https://mathoverflow.net/q/63183.

[Ham18] Joel David Hamkins. *Alan Turing, on computable numbers*. 2018. http://jdh.hamkins.org/alan-turing-on-computable-numbers.

[Ham21] Joel David Hamkins. *Lectures on the Philosophy of Mathematics*. MIT Press, 2021. ISBN: 9780262542234. https://mitpress.mit.edu/books/lectures-philosophy-mathematics.

[Lin03] Per Lindström. *Aspects of incompleteness*. Second. Vol. 10. Lecture Notes in Logic. Association for Symbolic Logic, Urbana, IL; A K Peters, Ltd., Natick, MA, 2003, pp. x+163. ISBN: 1-56881-173-X.

[Mon62] Richard Montague. "Theories incomparable with respect to relative interpretability". *J. Symbolic Logic* 27 (1962), pp. 195–211. ISSN: 0022-4812. DOI: 10.2307/2964114. https://ezproxy-prd.bodleian.ox.ac.uk:2102/10.2307/2964114.

[Tur36] A. M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society* 42.3 (1936), pp. 230–265. ISSN: 0024-6115. DOI: 10.1112/plms/s2-42.1.230.

[Wik18] Wikipedia. *Quine (computing)*. 2018. https://en.wikipedia.org/wiki/Quine_(computing) (version 01 September 2018).

(Joel David Hamkins) Professor of Logic, Oxford University & Sir Peter Strawson Fellow, University College, Oxford

Email address: joeldavid.hamkins@philosophy.ox.ac.uk

URL: http://jdh.hamkins.org