Self-reference in computability theory and the universal algorithm

Joel David Hamkins

City University of New York

CUNY Graduate Center

Mathematics, Philosophy, Computer Science

College of Staten Island

Mathematics

MathOverflow

Ouroboros: Formal Criteria of Self-Reference
in Mathematics and Philosophy
Bonn, February 16-18, 2018

Introduction

A delightful theorem

The universal algorithm

and its generalization to set theory,

The universal finite set.

Both rely fundamentally on self-reference.

Self-reference in computability

Definition

A *Quine* is a program that gives as output its own source code.

Imagine a Turing machine program that prints out its own source code on the tape.

Since this output can be used in further computation, a Quine is a program able to handle itself computationally.

Quine examples

Programmers compete to produce short Quines.

Quine examples

Programmers compete to produce short Quines.

Python Quine

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

Quine examples

Programmers compete to produce short Quines.

Python Quine

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

C Quine

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

Quine examples

Programmers compete to produce short Quines.

Python Quine

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

C Quine

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

SQL Quine

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

From the Quine page and StackOverflow.

Quines are not a computability concept

The Quine concept is not actually computability theoretic.

Quines are not a computability concept

The Quine concept is not actually computability theoretic.

Reason:

- It does not respect the equivalence of programs.
- 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 is no longer a Quine.

Nevertheless, Quines are fascinating!

Quines exist

Fact

Every Turing-complete programming language admits Quines.

Let me explain how this follows from the Kleene recursion theorem.

Recursion theorem: self-reference in computability

Kleene recursion theorem

For every computable transformation of the space of programs,

$$e \mapsto f(e)$$

there is a program e such that e and $f(e)$ compute the same function.

Recursion theorem: self-reference in computability

Kleene recursion theorem

For every computable transformation of the space of programs,

$$e \mapsto f(e)$$

there is a program e such that e and $f(e)$ compute the same function.

The program $f(e)$ can make explicit reference to e, and so the fixed point e in effect makes reference to itself.

The recursion theorem is a computability analogue of the Gödel-Carnap fixed-point lemma: for every $\varphi(x)$ in language of arithmetic, there is a sentence σ with

$$\sigma \leftrightarrow \varphi(\ulcorner \sigma \urcorner).$$

Quines exist in every Turing-complete language

Theorem

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

Quines exist in every Turing-complete language

Theorem

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 produce explicit Quines.

TEX is Turing-complete and therefore admits a Quine

TEX is Turing-complete and therefore admits a Quine

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
```

TEX is Turing-complete and therefore admits a Quine

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
```

Output:

```
\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
```

Ouroboros program: a Quine cycle

Ouroboros program: a Quine cycle

Java program

```
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]);

  }
}
```

Ouroboros program: a Quine cycle

Java program outputs a C++ program,

```java
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]);

  }
}
```

```cpp
#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;
}
```

Ouroboros program: a Quine cycle

Java program outputs a C++ program, which outputs the Java code.

```java
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]);

  }
}
```

```cpp
#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.

Universality via self-reference

Let me now explain how the capacity for self-reference in computability leads to universality.

Universal program (warm-up): the petulant child

Consider program e

It searches for a proof in PA of a statement:

"program e does not give output n."

When found, it gives output n and halts.

Universal program (warm-up): the petulant child

Consider program e

It searches for a proof in PA of a statement:

"program e does not give output n."

When found, it gives output n and halts.

(Use recursion theorem to eliminate the circularity.)

Universal program (warm-up): the petulant child

Consider program *e*

It searches for a proof in PA of a statement:

"program e does not give output n."

When found, it gives output *n* and halts.

(Use recursion theorem to eliminate the circularity.)

The program is a petulant child: upon finding a rule of forbidden behavior, it acts immediately to violate it.

Universal program (warm-up): the petulant child

Consider program *e*

It searches for a proof in PA of a statement:

"program e does not give output n."

When found, it gives output *n* and halts.

(Use recursion theorem to eliminate the circularity.)

The program is a petulant child: upon finding a rule of forbidden behavior, it acts immediately to violate it.

Key Observation: you cannot refute any particular output for this program, since then indeed it would have that output.

Universal program (warm-up): the petulant child

Consider program *e*

It searches for a proof in PA of a statement:

"program e does not give output n."

When found, it gives output *n* and halts.

(Use recursion theorem to eliminate the circularity.)

The program is a petulant child: upon finding a rule of forbidden behavior, it acts immediately to violate it.

Key Observation: you cannot refute any particular output for this program, since then indeed it would have that output.

Consequently, for any *n* it is consistent with PA that this program gives output *n*.

Universal program: the petulant child

Conclusion

There is a Turing machine program e, which we can write down, with the following properties.

Universal program: the petulant child

Conclusion

There is a Turing machine program e, which we can write down, with the following properties.

- When run in the standard model $\mathbb{N}$, the program never halts.

Universal program: the petulant child

Conclusion

There is a Turing machine program e, which we can write down, with the following properties.

- When run in the standard model $\mathbb{N}$, the program never halts.
- For any number n, there is a nonstandard model of arithmetic in which the program eventually gives output n.

Universal algorithm: sequence version

Consider program p

Searches for a proof of:

> *"program p does not enumerate the finite sequence $a_0, a_1, \ldots, a_n$ and then halt."*

When found, enumerate that sequence immediately.

Universal algorithm: sequence version

Consider program p

Searches for a proof of:

> *"program p does not enumerate the finite sequence $a_0, a_1, \ldots, a_n$ and then halt."*

When found, enumerate that sequence immediately.

You cannot refute any particular sequence being enumerated by p.

Universal algorithm: sequence version

Consider program p

Searches for a proof of:

> *"program p does not enumerate the finite sequence $a_0, a_1, \ldots, a_n$ and then halt."*

When found, enumerate that sequence immediately.

You cannot refute any particular sequence being enumerated by p.

So it is consistent with PA that p enumerates any desired sequence.

Universal program: function version

Theorem

There is a program p, such that for every function $f : \mathbb{N} \to \mathbb{N}$, there is a nonstandard model of arithmetic N, such that in N, program p on input $n \in \mathbb{N}$ gives output $f(n)$.

Universal program: function version

Theorem

There is a program p, such that for every function $f : \mathbb{N} \to \mathbb{N}$, there is a nonstandard model of arithmetic N, such that in N, program p on input $n \in \mathbb{N}$ gives output $f(n)$.

Every function can be computable! . . . in the right universe. And by the same universal program.

Universal program: function version

Theorem

There is a program p, such that for every function $f : \mathbb{N} \to \mathbb{N}$, there is a nonstandard model of arithmetic N, such that in N, program p on input $n \in \mathbb{N}$ gives output $f(n)$.

Every function can be computable! . . . in the right universe. And by the same universal program.

Proof.

The assertions

> *"the n^{th} number enumerated by p is $f(n)$"*

are finitely consistent with PA. So by compactness they are all true in some model. □

The universal algorithm: full extension version

Theorem (Woodin)

There is a Turing machine program e such that:

1. *Program e enumerates a finite sequence only, and* PA *proves this.*

The universal algorithm: full extension version

Theorem (Woodin)

There is a Turing machine program e such that:

1. *Program e enumerates a finite sequence only, and* PA *proves this.*
2. *Program e enumerates the empty sequence in the standard model* $\mathbb{N}$.

The universal algorithm: full extension version

Theorem (Woodin)

There is a Turing machine program e such that:

1. *Program e enumerates a finite sequence only, and* PA *proves this.*
2. *Program e enumerates the empty sequence in the standard model* $\mathbb{N}$.
3. *In any model of arithmetic, if e enumerates sequence s and t is an extension of s, then there is an end-extension of the model in which e enumerates t.*

The universal algorithm: full extension version

Theorem (Woodin)

There is a Turing machine program e such that:

1. *Program e enumerates a finite sequence only, and* PA *proves this.*
2. *Program e enumerates the empty sequence in the standard model* $\mathbb{N}$.
3. *In any model of arithmetic, if e enumerates sequence s and t is an extension of s, then there is an end-extension of the model in which e enumerates t.*

In particular, every finite sequence $s \in \mathbb{N}^{<\omega}$ is enumerated by e in some model $M \models \mathrm{PA}$.

History

- Woodin proved the theorem in 2011.

History

- Woodin proved the theorem in 2011.
- Rasmus Blanck and Ali Enayat generalized it and removed a restriction to countable models.

History

- Woodin proved the theorem in 2011.
- Rasmus Blanck and Ali Enayat generalized it and removed a restriction to countable models.
- Hamkins provided a simplified proof in 2017.

History

- Woodin proved the theorem in 2011.
- Rasmus Blanck and Ali Enayat generalized it and removed a restriction to countable models.
- Hamkins provided a simplified proof in 2017.
- Proof idea has affinity with the ‘exile’ argument in Solovay’s analysis of provability logic.

History

- Woodin proved the theorem in 2011.
- Rasmus Blanck and Ali Enayat generalized it and removed a restriction to countable models.
- Hamkins provided a simplified proof in 2017.
- Proof idea has affinity with the ‘exile’ argument in Solovay’s analysis of provability logic.
- Related to work of Shavrukov and Visser.

History

- Woodin proved the theorem in 2011.
- Rasmus Blanck and Ali Enayat generalized it and removed a restriction to countable models.
- Hamkins provided a simplified proof in 2017.
- Proof idea has affinity with the ‘exile’ argument in Solovay’s analysis of provability logic.
- Related to work of Shavrukov and Visser.
- Weaker forms go back to early 1960s, Mostowski and Kripke.

Proof

This is my proof.

Proof

This is my proof.

Let's define the universal algorithm e.

Proof

This is my proof.

Let's define the universal algorithm e.

- Proceed in stages, releasing the sequence in batches.
- Stage n succeeds, if there is a proof from fragment PA_k, smaller than fragments at earlier stages, of a statement

  "it is not the case that e has exactly n stages and releases s at stage n,"

  where s is an explicitly listed sequence of numbers.

Proof

This is my proof.

Let's define the universal algorithm e.

- Proceed in stages, releasing the sequence in batches.
- Stage n succeeds, if there is a proof from fragment PA_k, smaller than fragments at earlier stages, of a statement

  "it is not the case that e has exactly n stages and releases s at stage n,"

  where s is an explicitly listed sequence of numbers.
- In this case, release s at stage n. Proceed to next stage.

Proof of universal algorithm

Succinctly:

The program e enumerates s at stage n, if it finds proof, in a strictly smaller fragment of PA each time, that it does not do so as its last stage.

Proof of universal algorithm

Succinctly:

The program e enumerates s at stage n, if it finds proof, in a strictly smaller fragment of PA each time, that it does not do so as its last stage.

Thus, program e is a persistently petulant child

Upon finding a rule forbidding certain behavior, it immediately exhibits that behavior. But may act again, to violate a stronger rule.

Proof of universal algorithm

Succinctly:

The program e enumerates s at stage n, if it finds proof, in a strictly smaller fragment of PA each time, that it does not do so as its last stage.

Thus, program e is a persistently petulant child

Upon finding a rule forbidding certain behavior, it immediately exhibits that behavior. But may act again, to violate a stronger rule.

Self-reference

Use Kleene recursion theorem to find e, solving the circular definition.

Finiteness

Observation

The universal sequence is finite.

Finiteness

Observation

The universal sequence is finite.

Proof.

The successful fragments PA_k are getting smaller, so this can happen at most finitely many times.

Thus, the sequence enumerated by e will be finite.

And PA can undertake this argument. □

Empty in the standard model

Claim

If stage n is successful, then fragment PA_{k_n} is nonstandard.

Empty in the standard model

Claim

If stage n is successful, then fragment PA_{k_n} is nonstandard.

Proof.

Consider last stage n. The assertion

"e enumerates s with exactly n successful stages"

has complexity Σ^0_2. For standard k, the Mostowski reflection theorem shows $\mathrm{PA} \vdash \mathrm{Con}(\mathrm{Tr}_k)$.

So M cannot have proof from PA_k of something contrary to actual behavior. So k_n must be nonstandard. □

Empty in the standard model

Claim

If stage n is successful, then fragment PA_{k_n} is nonstandard.

Proof.

Consider last stage n. The assertion

"e enumerates s with exactly n successful stages"

has complexity Σ^0_2. For standard k, the Mostowski reflection theorem shows $\mathrm{PA} \vdash \mathrm{Con}(\mathrm{Tr}_k)$.

So M cannot have proof from PA_k of something contrary to actual behavior. So k_n must be nonstandard. □

In particular, e enumerates empty sequence in $\mathbb{N}$.

Proving the extension property of e

Assume e enumerates s in M and t is an extension of s.

Proving the extension property of e

Assume e enumerates s in M and t is an extension of s.

Let n be the first unsuccessful stage in M. Let k be nonstandard, smaller than earlier k_i.

Proving the extension property of e

Assume e enumerates s in M and t is an extension of s.

Let n be the first unsuccessful stage in M. Let k be nonstandard, smaller than earlier k_i.

Key Observation

Since stage n was not successful, M must think that

$$\mathrm{PA}_k + \text{“}e \text{ has exactly } n \text{ stages and enumerates } t\text{”}$$

is consistent.

Proving the extension property of e

Assume e enumerates s in M and t is an extension of s.

Let n be the first unsuccessful stage in M. Let k be nonstandard, smaller than earlier k_i.

Key Observation

Since stage n was not successful, M must think that

$$\mathrm{PA}_k + \text{“}e \text{ has exactly } n \text{ stages and enumerates } t\text{”}$$

is consistent.

So M can build a Henkin model of this theory, N. Note $N \models \mathrm{PA}$ since k is nonstandard.

Proving the extension property of e

Assume e enumerates s in M and t is an extension of s.

Let n be the first unsuccessful stage in M. Let k be nonstandard, smaller than earlier k_i.

Key Observation

Since stage n was not successful, M must think that

$$\mathrm{PA}_k + \text{“}e \text{ has exactly } n \text{ stages and enumerates } t\text{”}$$

is consistent.

So M can build a Henkin model of this theory, N. Note $N \models \mathrm{PA}$ since k is nonstandard.

So N end-extends M and thinks e enumerates t, as desired. □

It turns out that a number of classical results can be seen as immediate consequences of the universal algorithm.

Let us give a few of these examples.

Maximal Σ_1 diagrams

Corollary

The Σ_1 diagram of a model of arithmetic is never maximal.

Maximal Σ_1 diagrams

Corollary

The Σ_1 diagram of a model of arithmetic is never maximal.

Proof.

In any model of arithmetic M, there is some stage n that is not successful, but can become successful in an end-extension.

So the extension has new Σ_1 facts: "stage n is successful." □

Independent Π^0_1 sentences

Corollary (Kripke, Mostowski)

There are infinitely many independent Π^0_1 sentences

$$\eta_0,\ \eta_1,\ \eta_2, \ldots$$

Any desired true/false pattern is consistent with PA.

Independent Π^0_1 sentences

Corollary (Kripke, Mostowski)

There are infinitely many independent Π^0_1 sentences

$$\eta_0,\ \eta_1,\ \eta_2, \ldots$$

Any desired true/false pattern is consistent with PA.

Proof.

Let $\eta_k =$"k does not appear on the universal sequence." □

Independent buttons

Corollary "independent buttons"

There are Σ^0_1 sentences, all false in $\mathbb{N}$.

$$\rho_0,\ \rho_1,\ \rho_2,\ \cdots$$

But over any model of arithmetic M, any addition set of them I, coded in M, can become true in an end-extension N.

1. $N \models \rho_k$ for all $k \in I$.
2. For $k \notin I$, truth of ρ_k is not changed.

Independent buttons

Corollary "independent buttons"

There are Σ^0_1 sentences, all false in $\mathbb{N}$.

$$\rho_0,\ \rho_1,\ \rho_2,\ \cdots$$

But over any model of arithmetic M, any addition set of them I, coded in M, can become true in an end-extension N.

1. $N \models \rho_k$ for all $k \in I$.
2. For $k \notin I$, truth of ρ_k is not changed.

Proof.

Let $\rho_k =$"k appears on the universal sequence." □

Independent Orey sentences

Corollary "independent switches"

There is an infinite list of independent Orey sentences

$$\sigma_0,\ \sigma_1,\ \sigma_2,\ \ldots$$

Any pattern of truth I coded in M is realized in an end-extension.

$$N \models \sigma_k \qquad \longleftrightarrow \qquad k \in I.$$

Independent Orey sentences

Corollary "independent switches"

There is an infinite list of independent Orey sentences

$$\sigma_0,\ \sigma_1,\ \sigma_2,\ \ldots$$

Any pattern of truth I coded in M is realized in an end-extension.

$$N \models \sigma_k \qquad \longleftrightarrow \qquad k \in I.$$

Proof.

Let $\sigma_k =$"k^{th} binary bit of the last number on the universal sequence is 1." □

Flexible formulas

Corollary (Kripke)

There is a Σ_n formula $\varphi(x)$, for any $n \geq 2$, that can be made equivalent to any desired Σ_n formula $\phi(x)$ in an end-extension.

That is, for any $M \models \mathrm{PA}$ and ϕ, there is an end-extension satisfying

$$\forall x\, \varphi(x) \leftrightarrow \phi(x).$$

Flexible formulas

Corollary (Kripke)

There is a Σ_n formula $\varphi(x)$, for any $n \geq 2$, that can be made equivalent to any desired Σ_n formula $\phi(x)$ in an end-extension.

That is, for any $M \models \mathrm{PA}$ and ϕ, there is an end-extension satisfying

$$\forall x\, \varphi(x) \leftrightarrow \phi(x).$$

Proof.

Let $\varphi(x) = \Phi(k, x)$ where k is the last element of the universal sequence and Φ is a universal Σ_n formula. □

Set-theoretic analogue

What is the set-theoretic analogue of the universal algorithm?

One wants a version of the theorem for models of set theory.

Arithmetic vs. set theory

Arithmetic

The computably enumerable sets are gradually revealed as time proceeds. Elements are confirmed at some stage of time.

Arithmetic vs. set theory

Arithmetic

The computably enumerable sets are gradually revealed as time proceeds. Elements are confirmed at some stage of time.

Set theory

The locally verifiable sets have members confirmed in some V_α, as the set-theoretic universe grows.

$$\{ x \mid \varphi(x) \} \qquad \varphi(x) \leftrightarrow \exists \alpha \; V_\alpha \models \psi(x).$$

Arithmetic vs. set theory

Arithmetic

The computably enumerable sets are gradually revealed as time proceeds. Elements are confirmed at some stage of time.

Set theory

The locally verifiable sets have members confirmed in some V_α, as the set-theoretic universe grows.

$$\{ x \mid \varphi(x) \} \qquad \varphi(x) \leftrightarrow \exists \alpha \; V_\alpha \models \psi(x).$$

Elementary Fact

Locally verifiable sets $= \Sigma_2$ definable.

Question

So the set-theoretic analogue of c.e. is Σ_2 definable.

Question

So the set-theoretic analogue of c.e. is Σ_2 definable.

Question(Hamkins)

Is there a Σ_2 definable set $\{ x \mid \varphi(x) \}$ with the following?

- ZFC proves $\{ x \mid \varphi(x) \}$ is a set.
- For every countable model $M \models \mathrm{ZFC}$, if

$$M \models \{ x \mid \varphi(x) \} = y \subseteq z,$$

then there is top-extension N with

$$N \models \{ x \mid \varphi(x) \} = z.$$

Partial progress

I had initially made partial progress.

- I could do it with Π_3 definitions.
- Or with Σ_2, if you work only in models of certain theories, such as eventual GCH or $V \neq \mathrm{HOD}$.
- Or with Σ_2, if you allow $\{ x \mid \varphi(x) \}$ to sometimes be a proper class.

Woodin and I began to work together on the problem in September.

Universal finite set

The question is answered in joint work with W. Hugh Woodin.

Theorem (Hamkins + Woodin)

There is a Σ_2 definition φ such that

1. ZFC *proves* $\{ x \mid \varphi(x) \}$ *is finite.*

Universal finite set

The question is answered in joint work with W. Hugh Woodin.

Theorem (Hamkins + Woodin)

There is a Σ_2 definition φ such that

1. ZFC *proves* $\{ x \mid \varphi(x) \}$ *is finite.*
2. *If M is transitive, then $M \models \{ x \mid \varphi(x) \} = \emptyset$.*

Universal finite set

The question is answered in joint work with W. Hugh Woodin.

Theorem (Hamkins + Woodin)

There is a Σ_2 definition φ such that

1. ZFC *proves* $\{ x \mid \varphi(x) \}$ *is finite.*
2. *If M is transitive, then $M \models \{ x \mid \varphi(x) \} = \emptyset$.*
3. *If M is a countable model of* ZFC *with*

$$M \models \{ x \mid \varphi(x) \} = y \subseteq z,$$

where z is finite in M, then there is top-extension N with

$$N \models \{ x \mid \varphi(x) \} = z.$$

Universal countable set, universal set

The finite-set version of the theorem implies the other versions.

Universal countable set, universal set

The finite-set version of the theorem implies the other versions.

For example, the union of the universal finite set can be made an arbitrary set.

Universal countable set, universal set

The finite-set version of the theorem implies the other versions.

For example, the union of the universal finite set can be made an arbitrary set.

The union of the countable members of the universal finite set is an arbitrary countable set.

Universal countable set, universal set

The finite-set version of the theorem implies the other versions.

For example, the union of the universal finite set can be made an arbitrary set.

The union of the countable members of the universal finite set is an arbitrary countable set.

And so on for other cardinals or other kinds of universal sets.

Universal countable set, universal set

The finite-set version of the theorem implies the other versions.

For example, the union of the universal finite set can be made an arbitrary set.

The union of the countable members of the universal finite set is an arbitrary countable set.

And so on for other cardinals or other kinds of universal sets.

Also, there is a sequence version of the theorem.

Ideas from the proof

- The set is defined in stages.

Ideas from the proof

- The set is defined in stages.
- Look for absence of top-extensions of $\langle V_\beta, \in \rangle$, rather than for proofs.

Ideas from the proof

- The set is defined in stages.
- Look for absence of top-extensions of $\langle V_\beta, \in \rangle$, rather than for proofs.
- Sets are added at a stage if there is no suitable model in which they are added as the last elements.

Ideas from the proof

- The set is defined in stages.
- Look for absence of top-extensions of $\langle V_\beta, \in \rangle$, rather than for proofs.
- Sets are added at a stage if there is no suitable model in which they are added as the last elements.
- Self-reference via Gödel-Carnap fixed-point rather than Kleene recursion.

Ideas from the proof

- The set is defined in stages.
- Look for absence of top-extensions of $\langle V_\beta, \in \rangle$, rather than for proofs.
- Sets are added at a stage if there is no suitable model in which they are added as the last elements.
- Self-reference via Gödel-Carnap fixed-point rather than Kleene recursion.
- Handling the ω-standard model case involves forcing and rank of well-founded trees.

For a detailed argument, see the paper or my slides from the Prague Gathering, Beauty of Logic 2018 conference last month.

Applications of the universal finite set

Let me mention a few applications of the universal finite set theorem.

No model of set theory has maximal Σ_2 diagram

Theorem

No model of set theory M has a maximal Σ_2 diagram.

No model of set theory has maximal Σ_2 diagram

Theorem

No model of set theory M has a maximal Σ_2 diagram.

Proof.

If n is the first unsuccessful stage in M, then the assertion

"stage n is successful"

is Σ_2 assertion not yet true, but true in a top-extension of M. □

Maximal Σ_2 extensions of ZFC

Meanwhile, we can easily construct maximal consistent Σ_2 extensions of ZFC.

Maximal Σ_2 extensions of ZFC

Meanwhile, we can easily construct maximal consistent Σ_2 extensions of ZFC.

Any model of set theory with such a theory must have every standard stage n successful.

Maximal Σ_2 extensions of ZFC

Meanwhile, we can easily construct maximal consistent Σ_2 extensions of ZFC.

Any model of set theory with such a theory must have every standard stage n successful.

Corollary

No transitive model of ZFC has a maximal Σ_2 theory.

Maximal Σ_2 extensions of ZFC

Meanwhile, we can easily construct maximal consistent Σ_2 extensions of ZFC.

Any model of set theory with such a theory must have every standard stage n successful.

Corollary

No transitive model of ZFC has a maximal Σ_2 theory.

Implications for Maddy's MAXIMIZE in philosophy of set theory.

Maximal Σ_2 extensions of ZFC

Meanwhile, we can easily construct maximal consistent Σ_2 extensions of ZFC.

Any model of set theory with such a theory must have every standard stage n successful.

Corollary

No transitive model of ZFC has a maximal Σ_2 theory.

Implications for Maddy's MAXIMIZE in philosophy of set theory.

- Should maximize the sentences verifiable in some V_θ.
- But this never happens in well-founded models.
- Her principle therefore pushes us to ill-foundedness.

Σ_2 definability

Theorem

Every individual in a countable model of set theory M can become Σ_2 definable from a natural number parameter in a top-extension of M.

Indeed, there is a single definition and single parameter $n \in \omega^M$, such that every $a \in M$ is defined by that definition with that parameter in some top-extension N.

Σ_2 definability

Theorem

Every individual in a countable model of set theory M can become Σ_2 definable from a natural number parameter in a top-extension of M.

Indeed, there is a single definition and single parameter $n \in \omega^M$, such that every $a \in M$ is defined by that definition with that parameter in some top-extension N.

Proof.

The definition is, "the unique set added at stage n," where n is the first unsuccessful stage in M. □

Everybody is possibly necessarily special

Corollary

Every individual in a countable ω-standard model of set theory can become Σ_2 definable, without parameters, in a top-extension.

Everybody is possibly necessarily special

Corollary

Every individual in a countable ω-standard model of set theory can become Σ_2 definable, without parameters, in a top-extension.

Everybody can become special!

Everybody is possibly necessarily special

Corollary

Every individual in a countable ω-standard model of set theory can become Σ_2 definable, without parameters, in a top-extension.

Everybody can become special!

More precisely, in top-extensional potentialism, everybody is possibly necessarily special.

Everybody is possibly necessarily special

Corollary

Every individual in a countable ω-standard model of set theory can become Σ_2 definable, without parameters, in a top-extension.

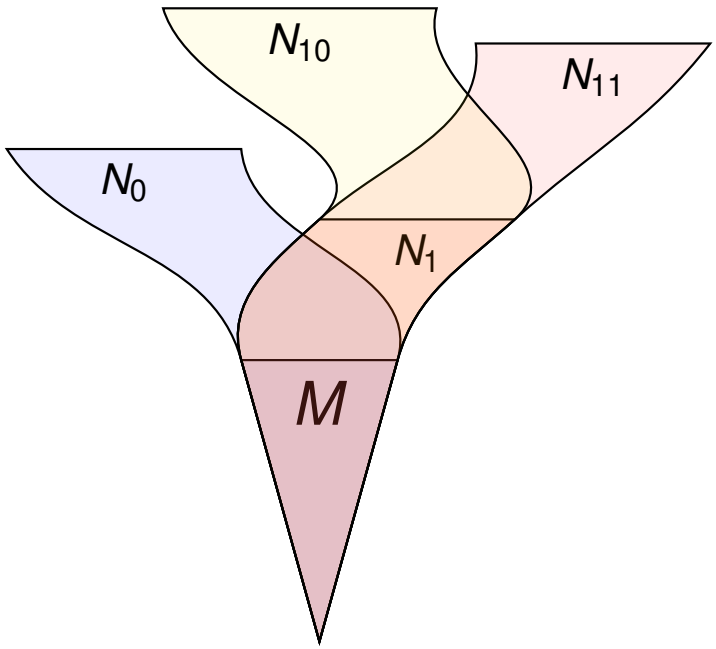
Everybody can become special!

More precisely, in top-extensional potentialism, everybody is possibly necessarily special.

Since N is also ω-standard, the result can be iterated.

The tree of top-extensions

This picture leads into the topic of potentialism, which I recently spoke about at Oxford and at the Winter School in Hejnice.



We analyze the precise modal validities of set-theoretic top-extensional potentialism and other concepts of potentialism.

References

Rasmus Blanck and Ali Enayat. “Marginalia on a theorem of Woodin”. *J. Symb. Log.* 82.1 (2017), pp. 359–374. ISSN: 0022-4812. DOI: 10.1017/jsl.2016.8.

Rasmus Blanck. “Contributions to the Metamathematics of Arithmetic. Fixed points, Independence, and Flexibility”. PhD thesis. University of Gothenburg, 2017. ISBN: 978-91-7346-917-3. http://hdl.handle.net/2077/52271.

Joel David Hamkins. “The modal logic of arithmetic potentialism and the universal algorithm”. *ArXiv e-prints* (2018). manuscript under review, pp. 1–35. arXiv:1801.04599[math.LO]. http://jdh.hamkins.org/arithmetic-potentialism-and-the-universal-algorithm.

Joel David Hamkins and W. Hugh Woodin. “The universal finite set”. *ArXiv e-prints* (2017). manuscript under review, pp. 1–16. arXiv:1711.07952[math.LO]. http://jdh.hamkins.org/the-universal-finite-set.

W. Hugh Woodin. “A potential subtlety concerning the distinction between determinism and nondeterminism”. In: *Infinity*. Cambridge Univ. Press, Cambridge, 2011, pp. 119–129.

Thank you.

Slides and articles available on http://jdh.hamkins.org.

Joel David Hamkins
City University of New York