Girard's Paradox: Impredicativity Rears its Ugly Head

Nathan Mull

University of Chicago
Department of Computer Science
October 19, 2022

Theory Lunch Talk

A Plan

- State Girard's paradox.
- Tell the story of how we get to Girard's paradox from the classical set-theoretic paradoxes.
- Describe why this is an interesting paradox, not *just* a set-theoretic analog.
- Throw a bit of type theory at you.

Disclaimer 1. This will be a bit of a departure. This isn't really TCS, maybe closer to logic or "technical philosophy." **So please stop me at any time for questions.**

Disclaimer 2. For those of you who do know a bit about what I'm talking about, I'm going to say some incorrect stuff. Feel free to grill me on it.

A Plan

- State Girard's paradox.
- Tell the story of how we get to Girard's paradox from the classical set-theoretic paradoxes.
- Describe why this is an interesting paradox, not *just* a set-theoretic analog.
- Throw a bit of type theory at you.

Disclaimer 1. This will be a bit of a departure. This isn't really TCS, maybe closer to logic or "technical philosophy." **So please stop me at any time for questions.**

Disclaimer 2. For those of you who do know a bit about what I'm talking about, I'm going to say some incorrect stuff. Feel free to grill me on it.

A Plan

- State Girard's paradox.
- Tell the story of how we get to Girard's paradox from the classical set-theoretic paradoxes.
- Describe why this is an interesting paradox, not *just* a set-theoretic analog.
- Throw a bit of type theory at you.

Disclaimer 1. This will be a bit of a departure. This isn't really TCS, maybe closer to logic or "technical philosophy." **So please stop me at any time for questions.**

Disclaimer 2. For those of you who do know a bit about what I'm talking about, I'm going to say some incorrect stuff. Feel free to grill me on it.

A Story

It's summer, 1902. The setting is the German city of Jena. We imagine a middle-aged Gottlob Frege—accomplished, perhaps weary—sitting in his home garden rereading a letter.

He is in preparations to publish his second volume on the logical foundations of arithmetic, albeit by less-than-ideal means; he could not find a publisher so he is paying for the printing himself.

The letter is from a young Bertrand Russell. It reveals an inconsistency in the logical system Frege had dedicated the last two decades working on.

(You can fill in the next part)

A Story

It's summer, 1902. The setting is the German city of Jena. We imagine a middle-aged Gottlob Frege—accomplished, perhaps weary—sitting in his home garden rereading a letter.

He is in preparations to publish his second volume on the logical foundations of arithmetic, albeit by less-than-ideal means; he could not find a publisher so he is paying for the printing himself.

The letter is from a young Bertrand Russell. It reveals an inconsistency in the logical system Frege had dedicated the last two decades working on.

(You can fill in the next part)

A Story

It's summer, 1902. The setting is the German city of Jena. We imagine a middle-aged Gottlob Frege—accomplished, perhaps weary—sitting in his home garden rereading a letter.

He is in preparations to publish his second volume on the logical foundations of arithmetic, albeit by less-than-ideal means; he could not find a publisher so he is paying for the printing himself.

The letter is from a young Bertrand Russell. It reveals an inconsistency in the logical system Frege had dedicated the last two decades working on.

(You can fill in the next part)

A Story

It's summer, 1902. The setting is the German city of Jena. We imagine a middle-aged Gottlob Frege—accomplished, perhaps weary—sitting in his home garden rereading a letter.

He is in preparations to publish his second volume on the logical foundations of arithmetic, albeit by less-than-ideal means; he could not find a publisher so he is paying for the printing himself.

The letter is from a young Bertrand Russell. It reveals an inconsistency in the logical system Frege had dedicated the last two decades working on.

(You can fill in the next part)

A Paradox

Comprehension Axiom Schema. $\exists X \forall x.(x \in X \Leftrightarrow \phi(x))$.

In human speak. Give me a statement about things, I can construct a set which contains the things that satisfy the statement.

Ex. If $\phi(x) = x \textit{ is red}$, then there is a set R of all red things, *i.e.*, $R = \{x \mid x \textit{ is red}\}$.

Ex. We can construct the set $\{X \mid X \in X\}$ (this turns out to be empty in most set theories).

Russell's Paradox. Consider $R = \{X \mid X \notin X\}$. *Is* $R \in R$*?*

A Paradox

Comprehension Axiom Schema. $\exists X \forall x.(x \in X \Leftrightarrow \phi(x))$.

In human speak. Give me a statement about things, I can construct a set which contains the things that satisfy the statement.

Ex. If $\phi(x) = x$ *is red*, then there is a set R of all red things, *i.e.*, $R = \{x \mid x \text{ is red}\}$.

Ex. We can construct the set $\{X \mid X \in X\}$ (this turns out to be empty in most set theories).

Russell's Paradox. Consider $R = \{X \mid X \notin X\}$. *Is* $R \in R$*?*

A Paradox

Comprehension Axiom Schema. $\exists X \forall x.(x \in X \Leftrightarrow \phi(x))$.

In human speak. Give me a statement about things, I can construct a set which contains the things that satisfy the statement.

Ex. If $\phi(x) = x$ *is red*, then there is a set R of all red things, *i.e.*, $R = \{x \mid x \textit{ is red}\}$.

Ex. We can construct the set $\{X \mid X \in X\}$ (this turns out to be empty in most set theories).

Russell's Paradox. Consider $R = \{X \mid X \notin X\}$. *Is* $R \in R$*?*

A Paradox

Comprehension Axiom Schema. $\exists X \forall x.(x \in X \Leftrightarrow \phi(x))$.

In human speak. Give me a statement about things, I can construct a set which contains the things that satisfy the statement.

Ex. If $\phi(x) = x$ *is red*, then there is a set R of all red things, *i.e.*, $R = \{x \mid x \textit{ is red}\}$.

Ex. We can construct the set $\{X \mid X \in X\}$ (this turns out to be empty in most set theories).

Russell's Paradox. Consider $R = \{X \mid X \notin X\}$. *Is* $R \in R$*?*

Ramified Type Theory (Russell, 1908)

Type Theory to the rescue (sort of).

Assign every object in the language a type (say, a natural number). Then we syntactically restrict $A \in B$ so that the type of A is less than the type of B.

We could never even write $X \in X$ since X has the same type as itself.

The takeaway. Type theory helps us avoid impredicativity, or self-referential definitions. It allows us to classify objects by complexity so that we know we're using less complex objects to build up more complex objects.

Ramified Type Theory (Russell, 1908)

Type Theory to the rescue (sort of).

Assign every object in the language a type (say, a natural number). Then we syntactically restrict $A \in B$ so that the type of A is less than the type of B.

We could never even write $X \in X$ since X has the same type as itself.

The takeaway. Type theory helps us avoid impredicativity, or self-referential definitions. It allows us to classify objects by complexity so that we know we're using less complex objects to build up more complex objects.

Ramified Type Theory (Russell, 1908)

Type Theory to the rescue (sort of).

Assign every object in the language a type (say, a natural number). Then we syntactically restrict $A \in B$ so that the type of A is less than the type of B.

We could never even write $X \in X$ since X has the same type as itself.

The takeaway. Type theory helps us avoid impredicativity, or self-referential definitions. It allows us to classify objects by complexity so that we know we're using less complex objects to build up more complex objects.

Ramified Type Theory (Russell, 1908)

Type Theory to the rescue (sort of).

Assign every object in the language a type (say, a natural number). Then we syntactically restrict $A \in B$ so that the type of A is less than the type of B.

We could never even write $X \in X$ since X has the same type as itself.

The takeaway. Type theory helps us avoid impredicativity, or self-referential definitions. It allows us to classify objects by complexity so that we know we're using less complex objects to build up more complex objects.

The set of all sets

Restricted Comprehension Axiom Schema.
$\forall Y \exists X \forall x.(x \in X \Leftrightarrow (\phi(x) \wedge x \in Y))$.

In human speak. Give me a statement about things, I can construct a set which contains the things **from a set I know exists** that satisfy the statement.

We can use Russell's paradox positively to prove there can be no set S of all sets. Otherwise, we could construct the Russell set

$$R = \{X \in S \mid X \notin X\}$$

The set of all sets

Restricted Comprehension Axiom Schema.
$\forall Y \exists X \forall x.(x \in X \Leftrightarrow (\phi(x) \wedge x \in Y))$.

In human speak. Give me a statement about things, I can construct a set which contains the things **from a set I know exists** that satisfy the statement.

We can use Russell's paradox positively to prove there can be no set S of all sets. Otherwise, we could construct the Russell set

$$R = \{X \in S \mid X \notin X\}$$

What is type theory?

Fair question…

To a Computer Scientist. A system for specifying the behavior of a program or function in a program. It makes programs more predictable and more easily composed.

To a Mathematician. A labeled rewrite system, where labels can be used to systematically describe a subset of the rewritable terms. Meta-theoretic questions can be then asked about this subset.

To a Philosopher. A way of defining ontological categories. We don't say "I read the car" because cars are not in the type of thing we read.

What is type theory?

Fair question…

To a Computer Scientist. A system for specifying the behavior of a program or function in a program. It makes programs more predictable and more easily composed.

To a Mathematician. A labeled rewrite system, where labels can be used to systematically describe a subset of the rewritable terms. Meta-theoretic questions can be then asked about this subset.

To a Philosopher. A way of defining ontological categories. We don't say "I read the car" because cars are not in the type of thing we read.

What is type theory?

Fair question…

To a Computer Scientist. A system for specifying the behavior of a program or function in a program. It makes programs more predictable and more easily composed.

To a Mathematician. A labeled rewrite system, where labels can be used to systematically describe a subset of the rewritable terms. Meta-theoretic questions can be then asked about this subset.

To a Philosopher. A way of defining ontological categories. We don't say "I read the car" because cars are not in the type of thing we read.

What is type theory?

Fair question…

To a Computer Scientist. A system for specifying the behavior of a program or function in a program. It makes programs more predictable and more easily composed.

To a Mathematician. A labeled rewrite system, where labels can be used to systematically describe a subset of the rewritable terms. Meta-theoretic questions can be then asked about this subset.

To a Philosopher. A way of defining ontological categories. We don't say "I read the car" because cars are not in the type of thing we read.

Curry-Howard Isomorphism

Question. What is type theory **to a logician**? Well, its a logic.

Some basic examples

```
id : a -> a
id x = x

tran : (a -> b) -> (b -> c) -> (a -> c)
tran f g x = f (g x)

double_neg : a -> ((a -> b) -> b)
double_neg x f = f x
```

Types *are* Theorems. Programs *are* proofs. Provability becomes type inhabitance.

Curry-Howard Isomorphism

Question. What is type theory **to a logician**? Well, its a logic.

Some basic examples

```
id : a -> a
id x = x

tran : (a -> b) -> (b -> c) -> (a -> c)
tran f g x = f (g x)

double_neg : a -> ((a -> b) -> b)
double_neg x f = f x
```

Types *are* Theorems. Programs *are* proofs. Provability becomes type inhabitance.

Curry-Howard Isomorphism

Question. What is type theory **to a logician**? Well, its a logic.

Some basic examples

```
id : a -> a
id x = x

tran : (a -> b) -> (b -> c) -> (a -> c)
tran f g x = f (g x)

double_neg : a -> ((a -> b) -> b)
double_neg x f = f x
```

Types *are* Theorems. Programs *are* proofs. Provability becomes type inhabitance.

Brouwer-Heyting-Kolmogorov Interpretation

The BHK interpretation describes a deep connection between proof and computation. It applies to a wide range of settings, but the main idea we want here:

View a proof of $A \to B$ as a *function* which maps a proof of A to a proof of B.

When we write a program of a given type, we're giving a compact representation of a formal proof tree.

Proofs are objects which can be directly manipulated.

Brouwer-Heyting-Kolmogorov Interpretation

The BHK interpretation describes a deep connection between proof and computation. It applies to a wide range of settings, but the main idea we want here:

View a proof of $A \to B$ as a *function* which maps a proof of A to a proof of B.

When we write a program of a given type, we're giving a compact representation of a formal proof tree.

Proofs are objects which can be directly manipulated.

Brouwer-Heyting-Kolmogorov Interpretation

The BHK interpretation describes a deep connection between proof and computation. It applies to a wide range of settings, but the main idea we want here:

View a proof of $A \to B$ as a *function* which maps a proof of A to a proof of B.

When we write a program of a given type, we're giving a compact representation of a formal proof tree.

Proofs are objects which can be directly manipulated.

Brouwer-Heyting-Kolmogorov Interpretation

The BHK interpretation describes a deep connection between proof and computation. It applies to a wide range of settings, but the main idea we want here:

View a proof of $A \to B$ as a *function* which maps a proof of A to a proof of B.

When we write a program of a given type, we're giving a compact representation of a formal proof tree.

Proofs are objects which can be directly manipulated.

Dependent Types (or the beginning of our woes)

Recap. Type are theorems. Programs are proofs. We write programs which manipulate proofs.

Question. But what about *actual* mathematics?

Suppose we want to prove (*i.e.*, implement) the theorem (*i.e.*, type) *all natural numbers have prime factorizations.*

This proof should be a *function* which maps n to a proof that n has a prime factorization. And should have a type that looks like

```
(x : Nat) -> has_prime_fact x
```

`has_prime_fact` has to *depend* on x. The proof that 2 has a factorization should not count as a proof that 100 has a factorization.

Dependent Types (or the beginning of our woes)

Recap. Type are theorems. Programs are proofs. We write programs which manipulate proofs.

Question. But what about *actual* mathematics?

Suppose we want to prove (*i.e.*, implement) the theorem (*i.e.*, type) *all natural numbers have prime factorizations.*

This proof should be a *function* which maps n to a proof that n has a prime factorization. And should have a type that looks like

```
(x : Nat) -> has_prime_fact x
```

`has_prime_fact` has to *depend* on x. The proof that 2 has a factorization should not count as a proof that 100 has a factorization.

Dependent Types (or the beginning of our woes)

Recap. Type are theorems. Programs are proofs. We write programs which manipulate proofs.

Question. But what about *actual* mathematics?

Suppose we want to prove (*i.e.*, implement) the theorem (*i.e.*, type) *all natural numbers have prime factorizations.*

This proof should be a *function* which maps n to a proof that n has a prime factorization. And should have a type that looks like

```
(x : Nat) -> has_prime_fact x
```

`has_prime_fact` has to *depend* on x. The proof that 2 has a factorization should not count as a proof that 100 has a factorization.

Dependent Types (or the beginning of our woes)

Recap. Type are theorems. Programs are proofs. We write programs which manipulate proofs.

Question. But what about *actual* mathematics?

Suppose we want to prove (*i.e.*, implement) the theorem (*i.e.*, type) *all natural numbers have prime factorizations.*

This proof should be a *function* which maps n to a proof that n has a prime factorization. And should have a type that looks like

```
(x : Nat) -> has_prime_fact x
```

`has_prime_fact` has to *depend* on x. The proof that 2 has a factorization should not count as a proof that 100 has a factorization.

Dependent Types (or the beginning of our woes)

Recap. Type are theorems. Programs are proofs. We write programs which manipulate proofs.

Question. But what about *actual* mathematics?

Suppose we want to prove (*i.e.*, implement) the theorem (*i.e.*, type) *all natural numbers have prime factorizations.*

This proof should be a *function* which maps n to a proof that n has a prime factorization. And should have a type that looks like

```
(x : Nat) -> has_prime_fact x
```

`has_prime_fact` has to *depend* on x. The proof that 2 has a factorization should not count as a proof that 100 has a factorization.

But wait...

Question. What is the type of `has_prime_fact`?

It should map a number to a *statement*. And we said theorems are types so lets say

```
has_prime_fact :: Nat -> Type
```

But wait (again)... what is the type of `Type`? Eh, let's say, `Type`.

But wait...

Question. What is the type of `has_prime_fact`?

It should map a number to a *statement*. And we said theorems are types so lets say

```
has_prime_fact :: Nat -> Type
```

But wait (again)... what is the type of `Type`? Eh, let's say, `Type`.

But wait...

Question. What is the type of `has_prime_fact`?

It should map a number to a *statement*. And we said theorems are types so lets say

```
has_prime_fact :: Nat -> Type
```

But wait (again)... what is the type of `Type`? Eh, let's say, `Type`.

What is type theory actually?

A type theory is specified by a grammar of terms (and types), and a collection of rules for deriving typing judgments.

Typing judgments are of the form

$$\Gamma \vdash M : A$$

which means M is of type A in the context Γ (a context is a collection of typed variables, think the environment in programming, or a collection of assumptions in logic)

The Principle of Explosion. A type theory is inconsistent if every type is inhabited, *i.e.*, for every A, there is a Γ and M such that $\Gamma \vdash M : A$.

What is type theory actually?

A type theory is specified by a grammar of terms (and types), and a collection of rules for deriving typing judgments.

Typing judgments are of the form

$$\Gamma \vdash M : A$$

which means M is of type A in the context Γ (a context is a collection of typed variables, think the environment in programming, or a collection of assumptions in logic)

The Principle of Explosion. A type theory is inconsistent if every type is inhabited, *i.e.*, for every A, there is a Γ and M such that $\Gamma \vdash M : A$.

What is type theory actually?

A type theory is specified by a grammar of terms (and types), and a collection of rules for deriving typing judgments.

Typing judgments are of the form

$$\Gamma \vdash M : A$$

which means M is of type A in the context Γ (a context is a collection of typed variables, think the environment in programming, or a collection of assumptions in logic)

The Principle of Explosion. A type theory is inconsistent if every type is inhabited, *i.e.*, for every A, there is a Γ and M such that $\Gamma \vdash M : A$.

Dependent Type Theory (Martin-Löf, 1972)

$$\frac{}{\emptyset \vdash} \qquad \frac{\Gamma \vdash A}{\Gamma,\, x : A \vdash} \qquad \frac{\Gamma \vdash \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B \qquad A \simeq_{\beta\eta} B}{\Gamma \vdash t : B}$$

$$\frac{\Gamma \vdash A \qquad \Gamma,\, x : A \vdash B}{\Gamma \vdash \Pi x : A.\, B} \qquad \frac{\Gamma,\, x : A \vdash t : B}{\Gamma \vdash \lambda x : A.\, t : \Pi x : A.\, B} \qquad \frac{\Gamma \vdash t : \Pi x : A.\, B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B\{x := u\}}$$

$$\frac{\Gamma \vdash A \qquad \Gamma,\, x : A \vdash B}{\Gamma \vdash \Sigma x : A.\, B} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B\{x := t\}}{\Gamma \vdash (t, u) : \Sigma x : A.\, B} \qquad \frac{\Gamma \vdash t : \Sigma x : A.\, B}{\Gamma \vdash \pi_1(t) : A}$$

$$\frac{\Gamma \vdash t : \Sigma x : A.\, B}{\Gamma \vdash \pi_2(t) : B\{x := \pi_1(t)\}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathbb{P}} \qquad \frac{\Gamma \vdash A : \mathbb{P}}{\Gamma \vdash \underline{A}} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{U}} \qquad \frac{\Gamma \vdash A : \mathcal{U}}{\Gamma \vdash \underline{A}}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash t \equiv_A u : \mathbb{P}} \qquad \frac{\Gamma \vdash t : A}{\Gamma \vdash \mathbf{refl}_{\equiv}\, t : \underline{t \equiv_A t}}$$

$$\frac{\Gamma,\, x : A,\, p : \underline{t \equiv_A x} \vdash P \qquad \Gamma \vdash q : \underline{t \equiv_A t'} \qquad \Gamma \vdash u : P\{x := t,\ p := \mathbf{refl}_{\equiv}\, t\}}{\Gamma \vdash J_{\equiv}(P,\, q,\, u) : P\{x := t',\ p := q\}}$$

$$(\lambda x : A.\, t)\, u \simeq_{\beta\eta} t\{x := u\} \qquad \lambda x : A.\, t\, x \simeq_{\beta\eta} t$$

$$\pi_1(t, t') \simeq_{\beta\eta} t \qquad \pi_2(t, t') \simeq_{\beta\eta} t' \qquad (\pi_1(t), \pi_2(t)) \simeq_{\beta\eta} t$$

$$J_{\equiv}(P,\ \mathbf{refl}_{\equiv}\, t,\ u) \simeq_{\beta\eta} u$$

Dependent Products

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.\, M : \Pi x^A.\, B}$$

$\lambda x^A.\, M$ is a function from A to B (where B might depend on x).

$\lambda x^A.\, M$ is a proof of the theorem "for all x of type A, $B(x)$ holds."

In code. `(fun x => M) : (x : A) -> B`

Dependent Products

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.\, M : \Pi x^A.\, B}$$

$\lambda x^A.\, M$ is a function from A to B (where B might depend on x).

$\lambda x^A.\, M$ is a proof of the theorem "for all x of type A, $B(x)$ holds."

In code. `(fun x => M) : (x : A) -> B`

Dependent Products

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A .\, M : \Pi x^A .\, B}$$

$\lambda x^A.\, M$ is a function from A to B (where B might depend on x).

$\lambda x^A.\, M$ is a proof of the theorem "for all x of type A, $B(x)$ holds."

In code. `(fun x => M) : (x : A) -> B`

Dependent Products

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A.\, M : \Pi x^A.\, B}$$

$\lambda x^A.\, M$ is a function from A to B (where B might depend on x).

$\lambda x^A.\, M$ is a proof of the theorem "for all x of type A, $B(x)$ holds."

In code. `(fun x => M) : (x :  A) -> B`

Dependent Sums

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B(M)}{\Gamma \vdash (M, N) : \sum x^A . B}$$

(M, N) is a pair where M is of type A and N is of type B instantiated at M.

(M, N) is a proof of the theorem "there exists an x of type A such that $B(x)$" with M as the witness and N as the proof that $B(M)$ holds.

In code. `(m, n) : (x : A ** B).`

Dependent Sums

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B(M)}{\Gamma \vdash (M, N) : \sum x^A . B}$$

(M, N) is a pair where M is of type A and N is of type B instantiated at M.

(M, N) is a proof of the theorem "there exists an x of type A such that $B(x)$" with M as the witness and N as the proof that $B(M)$ holds.

In code. `(m, n) : (x : A ** B).`

Dependent Sums

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B(M)}{\Gamma \vdash (M, N) : \sum x^A . B}$$

(M, N) is a pair where M is of type A and N is of type B instantiated at M.

(M, N) is a proof of the theorem "there exists an x of type A such that $B(x)$" with M as the witness and N as the proof that $B(M)$ holds.

In code. `(m, n) : (x : A ** B).`

Dependent Sums

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B(M)}{\Gamma \vdash (M, N) : \sum x^A . B}$$

(M, N) is a pair where M is of type A and N is of type B instantiated at M.

(M, N) is a proof of the theorem "there exists an x of type A such that $B(x)$" with M as the witness and N as the proof that $B(M)$ holds.

In code. `(m, n) :  (x :  A ** B).`

And of course $\vdash \mathsf{Type} : \mathsf{Type}$

Question. Why haven't we learned our lesson? Aren't we begging for a paradox?

Observation 1. Remember, we're writing programs, which can be used for "normal" computation as well. We can write functions can apply to numbers and functions that apply to proofs.

Observation 2. In set theory, we have two levels of discourse. One about sets and one about statements about sets. The statement '$x \in A$' is not a *part* of the set-theoretic universe, even though it can be represented in set theory.

In type theory, proofs and theorems have the same ontological status as objects like numbers, tree, groups, *etc*.

And of course $\vdash \mathsf{Type} : \mathsf{Type}$

Question. Why haven't we learned our lesson? Aren't we begging for a paradox?

Observation 1. Remember, we're writing programs, which can be used for "normal" computation as well. We can write functions can apply to numbers and functions that apply to proofs.

Observation 2. In set theory, we have two levels of discourse. One about sets and one about statements about sets. The statement '$x \in A$' is not a *part* of the set-theoretic universe, even though it can be represented in set theory.

In type theory, proofs and theorems have the same ontological status as objects like numbers, tree, groups, *etc*.

And of course $\vdash \mathsf{Type} : \mathsf{Type}$

Question. Why haven't we learned our lesson? Aren't we begging for a paradox?

Observation 1. Remember, we're writing programs, which can be used for "normal" computation as well. We can write functions can apply to numbers and functions that apply to proofs.

Observation 2. In set theory, we have two levels of discourse. One about sets and one about statements about sets. The statement '$x \in A$' is not a *part* of the set-theoretic universe, even though it can be represented in set theory.

In type theory, proofs and theorems have the same ontological status as objects like numbers, tree, groups, *etc*.

And of course $\vdash \mathsf{Type} : \mathsf{Type}$

Question. Why haven't we learned our lesson? Aren't we begging for a paradox?

Observation 1. Remember, we're writing programs, which can be used for "normal" computation as well. We can write functions can apply to numbers and functions that apply to proofs.

Observation 2. In set theory, we have two levels of discourse. One about sets and one about statements about sets. The statement '$x \in A$' is not a *part* of the set-theoretic universe, even though it can be represented in set theory.

In type theory, proofs and theorems have the same ontological status as objects like numbers, tree, groups, *etc*.

Russell's Paradox in Type Theory

Suppose we try to write the type

```
(A : Type ** (not (is_of_type A A)))
```

We can't write `is_of_type` inside type theory.

Comprehension allows the theory to affect things at the objects level which can lead to impredicativity.

We avoid this impredicativity by *internalizing* the theory, and then not allowing the *meta-theory* to play any role in the theory itself.

The catch. There are other set theoretic paradoxes! Ones more amenable to type-theoretic representation.

Russell's Paradox in Type Theory

Suppose we try to write the type

```
(A : Type ** (not (is_of_type A A)))
```

We can't write `is_of_type` inside type theory.

Comprehension allows the theory to affect things at the objects level which can lead to impredicativity.

We avoid this impredicativity by *internalizing* the theory, and then not allowing the *meta-theory* to play any role in the theory itself.

The catch. There are other set theoretic paradoxes! Ones more amenable to type-theoretic representation.

Russell's Paradox in Type Theory

Suppose we try to write the type

```
(A : Type ** (not (is_of_type A A)))
```

We can't write `is_of_type` inside type theory.

Comprehension allows the theory to affect things at the objects level which can lead to impredicativity.

We avoid this impredicativity by *internalizing* the theory, and then not allowing the *meta-theory* to play any role in the theory itself.

The catch. There are other set theoretic paradoxes! Ones more amenable to type-theoretic representation.

Russell's Paradox in Type Theory

Suppose we try to write the type

```
(A : Type ** (not (is_of_type A A)))
```

We can't write `is_of_type` inside type theory.

Comprehension allows the theory to affect things at the objects level which can lead to impredicativity.

We avoid this impredicativity by *internalizing* the theory, and then not allowing the *meta-theory* to play any role in the theory itself.

The catch. There are other set theoretic paradoxes! Ones more amenable to type-theoretic representation.

Russell's Paradox in Type Theory

Suppose we try to write the type

```
(A : Type ** (not (is_of_type A A)))
```

We can't write `is_of_type` inside type theory.

Comprehension allows the theory to affect things at the objects level which can lead to impredicativity.

We avoid this impredicativity by *internalizing* the theory, and then not allowing the *meta-theory* to play any role in the theory itself.

The catch. There are other set theoretic paradoxes! Ones more amenable to type-theoretic representation.

Other Set-Theoretic Paradoxes

Cantor's Paradox. There is no greatest cardinal number.

Burali-Forti Paradox. There is no set of all ordinal number.

Mirimanoff's Paradox. The set of well-founded sets is not well-founded, *i.e.*, there is no set of all well-founded sets.

Girard's Paradox (expressed in set theory). The strict well-quasi-ordering of all strict well-quasi-orderings is not well-founded.

Other Set-Theoretic Paradoxes

Cantor's Paradox. There is no greatest cardinal number.

Burali-Forti Paradox. There is no set of all ordinal number.

Mirimanoff's Paradox. The set of well-founded sets is not well-founded, *i.e.*, there is no set of all well-founded sets.

Girard's Paradox (expressed in set theory). The strict well-quasi-ordering of all strict well-quasi-orderings is not well-founded.

Other Set-Theoretic Paradoxes

Cantor's Paradox. There is no greatest cardinal number.

Burali-Forti Paradox. There is no set of all ordinal number.

Mirimanoff's Paradox. The set of well-founded sets is not well-founded, *i.e.*, there is no set of all well-founded sets.

Girard's Paradox (expressed in set theory). The strict well-quasi-ordering of all strict well-quasi-orderings is not well-founded.

Other Set-Theoretic Paradoxes

Cantor's Paradox. There is no greatest cardinal number.

Burali-Forti Paradox. There is no set of all ordinal number.

Mirimanoff's Paradox. The set of well-founded sets is not well-founded, *i.e.*, there is no set of all well-founded sets.

Girard's Paradox (expressed in set theory). The strict well-quasi-ordering of all strict well-quasi-orderings is not well-founded.

The Paradox in Naive Set Theory

A strict quasi-well-ordering is a set together with a transitive well-founded binary relation (no infinite descending sequences).

We can define the ordering $(X, <_X) <_\Omega (Y, <_Y)$ as: there exists a function $f : X \to Y$ which is bounded above (with respect to $<_Y$) and monotonic.

Lemma. $<_\Omega$ is transitive and well-founded, so we can define $(\Omega, <_\Omega)$, where Ω is the set of all strict quasi-well-orderings.

The final blow. $(\Omega, <_\Omega)$ is the *maximum* ordering. In particular, $(\Omega, <_\Omega) <_\Omega (\Omega, <_\Omega)$

The Paradox in Naive Set Theory

A strict quasi-well-ordering is a set together with a transitive well-founded binary relation (no infinite descending sequences).

We can define the ordering $(X, <_X) <_\Omega (Y, <_Y)$ as: there exists a function $f : X \to Y$ which is bounded above (with respect to $<_Y$) and monotonic.

Lemma. $<_\Omega$ is transitive and well-founded, so we can define $(\Omega, <_\Omega)$, where Ω is the set of all strict quasi-well-orderings.

The final blow. $(\Omega, <_\Omega)$ is the *maximum* ordering. In particular, $(\Omega, <_\Omega) <_\Omega (\Omega, <_\Omega)$

The Paradox in Naive Set Theory

A strict quasi-well-ordering is a set together with a transitive well-founded binary relation (no infinite descending sequences).

We can define the ordering $(X, <_X) <_\Omega (Y, <_Y)$ as: there exists a function $f : X \to Y$ which is bounded above (with respect to $<_Y$) and monotonic.

Lemma. $<_\Omega$ is transitive and well-founded, so we can define $(\Omega, <_\Omega)$, where Ω is the set of all strict quasi-well-orderings.

The final blow. $(\Omega, <_\Omega)$ is the *maximum* ordering. In particular, $(\Omega, <_\Omega) <_\Omega (\Omega, <_\Omega)$

The Paradox in Naive Set Theory

A strict quasi-well-ordering is a set together with a transitive well-founded binary relation (no infinite descending sequences).

We can define the ordering $(X, <_X) <_\Omega (Y, <_Y)$ as: there exists a function $f : X \to Y$ which is bounded above (with respect to $<_Y$) and monotonic.

Lemma. $<_\Omega$ is transitive and well-founded, so we can define $(\Omega, <_\Omega)$, where Ω is the set of all strict quasi-well-orderings.

The final blow. $(\Omega, <_\Omega)$ is the *maximum* ordering. In particular, $(\Omega, <_\Omega) <_\Omega (\Omega, <_\Omega)$

In Code (1/2)

```
tran : (a -> a -> Type) -> Type
tran {a} f = (x, y, z : a) -> f x y -> f y z -> f x z

nempty : (a -> Type) -> Type
nempty {a} p = (x : a ** p x)

no_lb : (a -> a -> Type) -> (a -> Type) -> Type
no_lb {a} f p = (x : a) -> p x
                        -> (y : a ** (p y, f y x))

wf : (a -> a -> Type) -> Type
wf {a} f = (p : a -> Type) -> nempty p
                           -> no_lb f p -> False
Omega : Type
Omega =
 (a : Type ** f : (a -> a -> Type) ** (tran f, wf f))
```

In Code (2/2)

```
Omega : Type
Omega =
 (a : Type ** f : (a -> a -> Type) ** (tran f, wf f))

LTN : Omega -> Omega -> Type
LTN (a ** ltn_a ** _) (b ** ltn_b ** _) =
    (  f : (a -> b)
    ** z : b
    ** ( (x, y : a) -> ltn_a x y -> ltn_b (f x) (f y)
       , (x : a) -> ltn_b (f x) z
       )
    )

Omega_as_Omega : Omega
Omega_as_Omega = (Omega ** LTN ** (..., ...))

Omega_LTN_Omega : LTN Omega_as_Omega Omega_as_Omega
Omega_LTN_Omega = ...
```

Girard's Paradox

Theorem. Martin-Löf's dependent type theory (as originally presented) is inconsistent.

In the code above, we can derive a term of type `False`, which is the same as the type `(A : Type) -> Type`.

The fix. Another heirarchy! We include

$$\mathsf{Type}_1 : \mathsf{Type}_2 : \mathsf{Type}_3 : \ldots$$

Girard's Paradox

Theorem. Martin-Löf's dependent type theory (as originally presented) is inconsistent.

In the code above, we can derive a term of type `False`, which is the same as the type `(A : Type) -> Type`.

The fix. Another heirarchy! We include

$$\mathsf{Type}_1 : \mathsf{Type}_2 : \mathsf{Type}_3 : \ldots$$

Girard's Paradox

Theorem. Martin-Löf's dependent type theory (as originally presented) is inconsistent.

In the code above, we can derive a term of type `False`, which is the same as the type `(A : Type) -> Type`.

The fix. Another heirarchy! We include

$$\mathsf{Type}_1 : \mathsf{Type}_2 : \mathsf{Type}_3 : \ldots$$

We don't need $\vdash$ Type : Type

Theorem. In fact, a much weaker system called λU is inconsistent *and this system doesn't have circular typing rules.*

We can even still play the entire game as before in the type hierarchy, but we can only derive

```
Omega_1 : Type_2
Omega_1 =
 (a : Type_1 ** f : (a -> a -> Type_1) ** (..., ...))

Omega_2 : Type_3
Omega_2 =
 (a : Type_2 ** f : (a -> a -> Type_1) ** (..., ...))

thm : LTE Omega_1_as_Omega Omega_2_as_Omega
...
```

So we can never derive the full contradiction.

We don't need $\vdash$ Type : Type

Theorem. In fact, a much weaker system called λU is inconsistent *and this system doesn't have circular typing rules.*

We can even still play the entire game as before in the type hierarchy, but we can only derive

```
Omega_1 : Type_2
Omega_1 =
 (a : Type_1 ** f : (a -> a -> Type_1) ** (..., ...))

Omega_2 : Type_3
Omega_2 =
 (a : Type_2 ** f : (a -> a -> Type_1) ** (..., ...))

thm : LTE Omega_1_as_Omega Omega_2_as_Omega
...
```

So we can never derive the full contradiction.

Final Remarks

Despite this being a very old question there is still a lot that is not known, and our modern perspective of type theory might allow us to approach these questions more readily.

Open Questions.

- Are the any other systems besides λU that are inconsistent?
- Can all set-theoretic paradoxes eventually be translated into type theory?
- If a system has a non-normalizing term (an infinite loop), is it inconsistent?
- Does inconsistency always imply a fixed-point combinator?

`https://github.com/nmmull/Falsum`

Final Remarks

Despite this being a very old question there is still a lot that is not known, and our modern perspective of type theory might allow us to approach these questions more readily.

Open Questions.

- Are the any other systems besides λU that are inconsistent?
- Can all set-theoretic paradoxes eventually be translated into type theory?
- If a system has a non-normalizing term (an infinite loop), is it inconsistent?
- Does inconsistency always imply a fixed-point combinator?

https://github.com/nmmull/Falsum

Final Remarks

Despite this being a very old question there is still a lot that is not known, and our modern perspective of type theory might allow us to approach these questions more readily.

Open Questions.

- Are the any other systems besides λU that are inconsistent?
- Can all set-theoretic paradoxes eventually be translated into type theory?
- If a system has a non-normalizing term (an infinite loop), is it inconsistent?
- Does inconsistency always imply a fixed-point combinator?

`https://github.com/nmmull/Falsum`