Universes and Univalence

Niels van der Weide

6 December, 2022

Universes

Univalence

Universe types

Goal: we want to have a type whose inhabitants are types
Such types are called **universes**

Universe types

Goal: we want to have a type whose inhabitants are types
Such types are called **universes**
Approaches:

- We could say Type : Type. However, this is inconsistent (Girard's paradox).

Universe types

Goal: we want to have a type whose inhabitants are types
Such types are called **universes**
Approaches:

- We could say Type : Type. However, this is inconsistent (Girard's paradox).
- Tarski style universes. We have a type $\mathcal{U}$ whose inhabitants are codes for types. We also have a operation that assigns to each $A : \mathcal{U}$ a type $\mathsf{El}(A)$.
  Tarski style universes are easier semantically, but more difficult to use.

Universe types

Goal: we want to have a type whose inhabitants are types
Such types are called **universes**
Approaches:

- We could say Type : Type. However, this is inconsistent (Girard's paradox).
- Tarski style universes. We have a type $\mathcal{U}$ whose inhabitants are codes for types. We also have a operation that assigns to each $A : \mathcal{U}$ a type $\mathsf{El}(A)$.
  Tarski style universes are easier semantically, but more difficult to use.
- **Russel style universes**. We have a type $\mathcal{U}$ whose inhabitants types. So, if we have $A : \mathcal{U}$, then A is a type. This $\mathcal{U}$ is closed under some type formers (products, sums, dependent products, and so on).

Russel style universes

Formation:

$$\Gamma \vdash \mathcal{U} : \mathsf{Type}$$

Introduction rules:

$$\Gamma \vdash \mathbf{0} : \mathcal{U}$$

$$\Gamma \vdash \mathbf{1} : \mathcal{U}$$

Russel style universes

More introduction rules:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash A \times B : \mathcal{U}}$$

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash A + B : \mathcal{U}}$$

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma \vdash B : \mathcal{U}}{\Gamma \vdash A \to B : \mathcal{U}}$$

Russel style universes

Even more introduction rules:

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A \vdash B : \mathcal{U}}{\Gamma \vdash \prod(x : A).B : \mathcal{U}}$$

$$\frac{\Gamma \vdash A : \mathcal{U} \qquad \Gamma, x : A \vdash B : \mathcal{U}}{\Gamma \vdash \sum(x : A).B : \mathcal{U}}$$

Universe Hierarchy

- Note: we cannot have $\mathcal{U} : \mathcal{U}$.
- In order to assign a type to $\mathcal{U}$, we add more universes
- So, we have universes $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \ldots$

We add the following introduction rule

$$\Gamma \vdash \mathcal{U}_n : \mathcal{U}_{n+1}$$

Cumulativity

We also require our universes to be **cumulative**. This means

$$\frac{\Gamma \vdash A : \mathcal{U}_n \qquad n \leq m}{\Gamma \vdash A : \mathcal{U}_m}$$

Type formers

Up to now, we introduced the following type formers

- The empty type: $\mathbf{0}$
- The unit type: $\mathbf{1}$
- Product types: $A \times B$
- Sum types: $A + B$
- Function types: $A \to B$
- Dependent products (for all): $\prod(x : A).B$
- Dependent sums (there is): $\sum(x : A).B$
- Universes: $\mathcal{U}$

Question: can we classify the identity types of these type formers?

Equality in type formers

Last week, we saw:

- For all $x, y : \mathbf{1}$, we have an equivalence $(x = y) \simeq \mathbf{1}$. Here the map $(x = y) \to \mathbf{1}$ is given by

$$\lambda(p : x = y).\star$$

Equality in type formers

Last week, we saw:

- For all $x, y : \mathbf{1}$, we have an equivalence $(x = y) \simeq \mathbf{1}$. Here the map $(x = y) \to \mathbf{1}$ is given by

$$\lambda(p : x = y).\star$$

- For all $x, y : A \times B$, we have an equivalence

$$(x = y) \simeq (\pi_1\, x = \pi_1\, y) \times (\pi_2\, x = \pi_2\, y).$$

given by

$$\lambda(p : x = y).\langle \mathsf{ap}\ \pi_1\ p, \mathsf{ap}\ \pi_2\ p\rangle$$

Equality in type formers

- The empty type:
- The unit type: ✓
- Product types: ✓
- Sum types:
- Function types:
- Dependent products (for all):
- Dependent sums (there is):
- Universes:

More equality

- **Empty type**: for $x, y : \mathbf{0}$, we have $(x = y) \simeq \mathbf{1}$.
- **Dependent sums**: we can also characterize equality for dependent sums (2.7), but that is more difficult

Equality in type formers

- The empty type: $\checkmark$
- The unit type: $\checkmark$
- Product types: $\checkmark$
- Sum types:
- Function types:
- Dependent products (for all):
- Dependent sums (there is): $\checkmark$
- Universes:

Equality in sum types

Question: suppose, $x, y : A + B$. How can we characterize $x = y$?

Equality in sum types

Question: suppose, $x, y : A + B$. How can we characterize $x = y$? Well, there are 4 cases

- $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inl}\ a_2$
- $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inr}\ b_2$
- $x \equiv \mathsf{inr}\ b_1$ and $y \equiv \mathsf{inl}\ a_2$
- $x \equiv \mathsf{inr}\ b_1$ and $y \equiv \mathsf{inr}\ b_2$

Equality in sum types: codes

Define a type $\mathsf{code} : A + B \to A + B \to \mathcal{U}$ as follows

$$\mathsf{code}\,(\mathsf{inl}\ a_1)\,(\mathsf{inl}\ a_2) \equiv (a_1 = a_2)$$

$$\mathsf{code}\,(\mathsf{inl}\ a_1)\,(\mathsf{inr}\ b_2) \equiv \mathbf{0}$$

$$\mathsf{code}\,(\mathsf{inr}\ b_1)\,(\mathsf{inl}\ a_2) \equiv \mathbf{0}$$

$$\mathsf{code}\,(\mathsf{inr}\ b_1)\,(\mathsf{inr}\ b_2) \equiv (b_1 = b_2)$$

These codes represents when elements are equal

Equality in sum types: codes

Define a type $\mathsf{code} : A + B \to A + B \to \mathcal{U}$ as follows

$$\mathsf{code}\,(\mathsf{inl}\ a_1)\,(\mathsf{inl}\ a_2) \equiv (a_1 = a_2)$$

$$\mathsf{code}\,(\mathsf{inl}\ a_1)\,(\mathsf{inr}\ b_2) \equiv \mathbf{0}$$

$$\mathsf{code}\,(\mathsf{inr}\ b_1)\,(\mathsf{inl}\ a_2) \equiv \mathbf{0}$$

$$\mathsf{code}\,(\mathsf{inr}\ b_1)\,(\mathsf{inr}\ b_2) \equiv (b_1 = b_2)$$

These codes represents when elements are equal
Goal: for all $x, y : A + B$, we have $(x = y) \simeq \mathsf{code}\ x\ y$.

Equality in sum types: encoding

We construct $\mathsf{encode} : (x = y) \to \mathsf{code}\ x\ y$.

- We use path induction

Equality in sum types: encoding

We construct encode : $(x = y) \to \mathsf{code}\ x\ y$.

- We use path induction
- So, we need to construct an inhabitant of code $x\,x$.

Equality in sum types: encoding

We construct $\mathsf{encode} : (x = y) \to \mathsf{code}\ x\ y$.

- We use path induction
- So, we need to construct an inhabitant of $\mathsf{code}\ x\ x$.
- We use case distinction on x.

Equality in sum types: encoding

We construct $\mathsf{encode} : (x = y) \to \mathsf{code}\ x\ y$.

- We use path induction
- So, we need to construct an inhabitant of $\mathsf{code}\ x\ x$.
- We use case distinction on x.
- If $x \equiv \mathsf{inl}\ a$, we construct an element of $a = a$. Take refl_a

Equality in sum types: encoding

We construct $\mathsf{encode} : (x = y) \to \mathsf{code}\ x\ y$.

- We use path induction
- So, we need to construct an inhabitant of $\mathsf{code}\ x\ x$.
- We use case distinction on x.
- If $x \equiv \mathsf{inl}\ a$, we construct an element of $a = a$. Take refl_a
- If $x \equiv \mathsf{inr}\ b$, we construct an element of $b = b$. Take refl_b

Equality in sum types: encoding

We construct encode : $(x = y) \to \mathsf{code}\; x\; y$.

- We use path induction
- So, we need to construct an inhabitant of code $x\,x$.
- We use case distinction on x.
- If $x \equiv \mathsf{inl}\; a$, we construct an element of $a = a$. Take refl_a
- If $x \equiv \mathsf{inr}\; b$, we construct an element of $b = b$. Take refl_b

Core step: the codes are reflexive

Equality in sum types: decoding

We construct decode : code $x\,y \to (x = y)$.

- We use case distinction x and y

Equality in sum types: decoding

We construct $\mathsf{decode} : \mathsf{code}\ x\ y \to (x = y)$.

- We use case distinction x and y
- If $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inl}\ a_2$, then we need to construct $(a_1 = a_2) \to \mathsf{inl}\ a_1 = \mathsf{inl}\ a_2$.
  Take $\lambda(p : a_1 = a_2).\, \mathsf{ap}\ \mathsf{inl}\ p$

Equality in sum types: decoding

We construct decode : code $x\, y \to (x = y)$.

- We use case distinction x and y
- If $x \equiv \text{inl } a_1$ and $y \equiv \text{inl } a_2$, then we need to construct $(a_1 = a_2) \to \text{inl } a_1 = \text{inl } a_2$.
  Take $\lambda(p : a_1 = a_2).\, \text{ap inl } p$
- If $x \equiv \text{inl } a_1$ and $y \equiv \text{inr } b_2$, then we need to construct $\mathbf{0} \to \text{inl } a_1 = \text{inr } b_2$. Empty elimination

Equality in sum types: decoding

We construct $\mathsf{decode} : \mathsf{code}\ x\ y \to (x = y)$.

- We use case distinction x and y
- If $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inl}\ a_2$, then we need to construct $(a_1 = a_2) \to \mathsf{inl}\ a_1 = \mathsf{inl}\ a_2$.
  Take $\lambda(p : a_1 = a_2).\, \mathsf{ap}\ \mathsf{inl}\ p$
- If $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inr}\ b_2$, then we need to construct $\mathbf{0} \to \mathsf{inl}\ a_1 = \mathsf{inr}\ b_2$. Empty elimination
- If $x \equiv \mathsf{inr}\ b_1$ and $y \equiv \mathsf{inl}\ a_2$, then we need to construct $\mathbf{0} \to \mathsf{inr}\ b_1 = \mathsf{inl}\ a_2$. Empty elimination

Equality in sum types: decoding

We construct decode : code $x\,y \to (x = y)$.

- We use case distinction x and y
- If $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inl}\ a_2$, then we need to construct $(a_1 = a_2) \to \mathsf{inl}\ a_1 = \mathsf{inl}\ a_2$.
  Take $\lambda(p : a_1 = a_2).\, \mathsf{ap}\ \mathsf{inl}\ p$
- If $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inr}\ b_2$, then we need to construct $\mathbf{0} \to \mathsf{inl}\ a_1 = \mathsf{inr}\ b_2$. Empty elimination
- If $x \equiv \mathsf{inr}\ b_1$ and $y \equiv \mathsf{inl}\ a_2$, then we need to construct $\mathbf{0} \to \mathsf{inr}\ b_1 = \mathsf{inl}\ a_2$. Empty elimination
- If $x \equiv \mathsf{inr}\ b_1$ and $y \equiv \mathsf{inr}\ b_2$, then we need to construct $(b_1 = b_2) \to \mathsf{inr}\ b_1 = \mathsf{inr}\ b_2$.
  Take $\lambda(p : a_1 = a_2).\, \mathsf{ap}\ \mathsf{inl}\ p$

Equality in sum types: decoding

We construct decode : code $x\,y \to (x = y)$.

- We use case distinction x and y
- If $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inl}\ a_2$, then we need to construct $(a_1 = a_2) \to \mathsf{inl}\ a_1 = \mathsf{inl}\ a_2$.
  Take $\lambda(p : a_1 = a_2).\, \mathsf{ap}\ \mathsf{inl}\ p$
- If $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inr}\ b_2$, then we need to construct $\mathbf{0} \to \mathsf{inl}\ a_1 = \mathsf{inr}\ b_2$. Empty elimination
- If $x \equiv \mathsf{inr}\ b_1$ and $y \equiv \mathsf{inl}\ a_2$, then we need to construct $\mathbf{0} \to \mathsf{inr}\ b_1 = \mathsf{inl}\ a_2$. Empty elimination
- If $x \equiv \mathsf{inr}\ b_1$ and $y \equiv \mathsf{inr}\ b_2$, then we need to construct $(b_1 = b_2) \to \mathsf{inr}\ b_1 = \mathsf{inr}\ b_2$.
  Take $\lambda(p : a_1 = a_2).\, \mathsf{ap}\ \mathsf{inl}\ p$

Choose the codes so that they imply equality

Equality in sum types: encode decode

We show: $\mathrm{decode}(\mathrm{encode}(p)) = p$ for $p : x = y$.

- We use path induction, so we need to show $\mathrm{decode}(\mathrm{encode}(\mathrm{refl}_x)) = \mathrm{refl}_x$

Equality in sum types: encode decode

We show: $\mathrm{decode}(\mathrm{encode}(p)) = p$ for $p : x = y$.

- We use path induction, so we need to show $\mathrm{decode}(\mathrm{encode}(\mathrm{refl}_x)) = \mathrm{refl}_x$
- Use case distinction on x

Equality in sum types: encode decode

We show: $\mathsf{decode}(\mathsf{encode}(p)) = p$ for $p : x = y$.

- We use path induction, so we need to show $\mathsf{decode}(\mathsf{encode}(\mathsf{refl}_x)) = \mathsf{refl}_x$
- Use case distinction on x
- Suppose $x \equiv \mathsf{inl}\ a$ (the other case is similar).

Equality in sum types: encode decode

We show: $\mathsf{decode}(\mathsf{encode}(p)) = p$ for $p : x = y$.

- We use path induction, so we need to show $\mathsf{decode}(\mathsf{encode}(\mathsf{refl}_x)) = \mathsf{refl}_x$
- Use case distinction on x
- Suppose $x \equiv \mathsf{inl}\ a$ (the other case is similar).
- Then $\mathsf{encode}(\mathsf{refl}_x)$ simplifies to refl_a.

Equality in sum types: encode decode

We show: $\mathsf{decode}(\mathsf{encode}(p)) = p$ for $p : x = y$.

- We use path induction, so we need to show $\mathsf{decode}(\mathsf{encode}(\mathsf{refl}_x)) = \mathsf{refl}_x$
- Use case distinction on x
- Suppose $x \equiv \mathsf{inl}\ a$ (the other case is similar).
- Then $\mathsf{encode}(\mathsf{refl}_x)$ simplifies to refl_a.
- Then $\mathsf{decode}(\mathsf{refl}_a) \equiv \mathsf{ap}\ \mathsf{inl}\ \mathsf{refl}_a$, which simplifies to $\mathsf{refl}_{\mathsf{inl}\ a}$.

Equality in sum types: decode encode, case 1

We show: $\mathrm{encode}(\mathrm{decode}(p)) = p$ for $p : \mathrm{code}\ x\ y$.

- Use case distinction on x and y

Equality in sum types: decode encode, case 1

We show: $\mathsf{encode}(\mathsf{decode}(p)) = p$ for $p : \mathsf{code}\ x\ y$.

- Use case distinction on x and y
- Case 1: $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inl}\ a_2$. Then $p : a_1 = a_2$

Equality in sum types: decode encode, case 1

We show: $\mathsf{encode}(\mathsf{decode}(p)) = p$ for $p : \mathsf{code}\ x\ y$.

- Use case distinction on x and y
- Case 1: $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inl}\ a_2$. Then $p : a_1 = a_2$
- Use path induction on p.

Equality in sum types: decode encode, case 1

We show: $\mathsf{encode}(\mathsf{decode}(p)) = p$ for $p : \mathsf{code}\ x\ y$.

- Use case distinction on x and y
- Case 1: $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inl}\ a_2$. Then $p : a_1 = a_2$
- Use path induction on p.
- Note $\mathsf{decode}(\mathsf{refl}_{a_1}) \equiv \mathsf{ap}\ \mathsf{inl}\ \mathsf{refl}_{a_1} \equiv \mathsf{refl}_{\mathsf{inl}\ a_1}$.

Equality in sum types: decode encode, case 1

We show: $\mathrm{encode}(\mathrm{decode}(p)) = p$ for $p : \mathrm{code}\ x\ y$.

- Use case distinction on x and y
- Case 1: $x \equiv \mathrm{inl}\ a_1$ and $y \equiv \mathrm{inl}\ a_2$. Then $p : a_1 = a_2$
- Use path induction on p.
- Note $\mathrm{decode}(\mathrm{refl}_{a_1}) \equiv \mathrm{ap}\ \mathrm{inl}\ \mathrm{refl}_{a_1} \equiv \mathrm{refl}_{\mathrm{inl}\ a_1}$.
- By definition $\mathrm{encode}\,\mathrm{refl}_{\mathrm{inl}\ a_1} \equiv \mathrm{refl}_{a_1}$.

The case $x \equiv \mathrm{inr}\ b_1$ and $y \equiv \mathrm{inr}\ b_2$ is similar.

Equality in sum types: decode encode, case 1

- Case 2: $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inr}\ b_2$. Then $p : \mathbf{0}$

Equality in sum types: decode encode, case 1

- Case 2: $x \equiv \mathsf{inl}\ a_1$ and $y \equiv \mathsf{inr}\ b_2$. Then $p : \mathbf{0}$
- Use empty elimination (ex falso quodlibit) on p

The case $x \equiv \mathsf{inr}\ b_1$ and $y \equiv \mathsf{inl}\ a_2$ is similar.

Encode-decode method

- The approach we used in the proof, was the so-called **encode-decode method**.
- This approach can be used for other types as well, such as the natural numbers.

Encode-decode method

- The approach we used in the proof, was the so-called **encode-decode method**.
- This approach can be used for other types as well, such as the natural numbers.
- Really, this stuff is nicer to do in a proof assistant, because the proof assistant does the bureaucracy for you, and nobody likes writing a large amount of bureaucratic steps on slides.

Equality in type formers

- The empty type: $\checkmark$
- The unit type: $\checkmark$
- Product types: $\checkmark$
- Sum types: $\checkmark$
- Function types:
- Dependent products (for all):
- Dependent sums (there is): $\checkmark$
- Universes:

Function extensionality

Suppose, we have $f, g : A \to B$. How to show $f = g$?

- We would like to have: if for all $a : A$ we have $f\ a = g\ a$, then $f = g$
- Formal notation:

$$\text{funext} : (\prod(a : A).f\ a = g\ a) \to f = g$$

- This is called **function extensionality**
- It means: if two functions have the same behavior, then they are equal irregardless of their implementation

Function extensionality (improved)

- Recall: we are characterizing $f = g$ up to equivalence
- As such, it is not sufficient to just have a map

$$\mathsf{funext} : \left(\prod(a : A).f\ a = g\ a\right) \to f = g$$

Function extensionality (improved)

- Recall: we are characterizing $f = g$ up to equivalence
- As such, it is not sufficient to just have a map

$$\mathsf{funext} : \left(\prod(a : A). f\ a = g\ a\right) \to f = g$$

- Instead, we define a map using path induction

$$\mathsf{happly} : f = g \to \left(\prod(a : A). f\ a = g\ a\right)$$

- Function extensionality says that happly is an equivalence

Function extensionality (improved)

- Recall: we are characterizing $f = g$ up to equivalence
- As such, it is not sufficient to just have a map

$$\text{funext} : (\prod(a : A).f\ a = g\ a) \to f = g$$

- Instead, we define a map using path induction

$$\text{happly} : f = g \to (\prod(a : A).f\ a = g\ a)$$

- Function extensionality says that happly is an equivalence
- From this, we get funext, but we also get that $\text{happly}(\text{funext}(p)) = p$ and $\text{funext}(\text{happly}(p)) = p$.

Equality in type formers

- The empty type: ✓
- The unit type: ✓
- Product types: ✓
- Sum types: ✓
- Function types: **function extensionality**
- Dependent products (for all): **function extensionality**
- Dependent sums (there is): ✓
- Universes:

The univalence axiom

Let $A, B : \mathcal{U}$. When do we have $A = B$?

- **Note**: we cannot say the condition $x : A$ if and only if $x : B$!

The univalence axiom

Let $A, B : \mathcal{U}$. When do we have $A = B$?

- **Note**: we cannot say the condition $x : A$ if and only if $x : B$!
- Instead we use equivalences

The univalence axiom

Let $A, B : \mathcal{U}$. When do we have $A = B$?

- **Note**: we cannot say the condition $x : A$ if and only if $x : B$!
- Instead we use equivalences
- **Note**: we have a map $\mathrm{idtoeqv} : (A = B) \to (A \simeq B)$.
- **Univalence**: the map idtoeqv is an equivalence of types

The univalence axiom

Let $A, B : \mathcal{U}$. When do we have $A = B$?

- **Note**: we cannot say the condition $x : A$ if and only if $x : B$!
- Instead we use equivalences
- **Note**: we have a map $\mathrm{idtoeqv} : (A = B) \to (A \simeq B)$.
- **Univalence**: the map idtoeqv is an equivalence of types
- From univalence, we get a map $\mathrm{ua} : (A \simeq B) \to (A = B)$

More on univalence

Let's prove $\mathsf{Bool} = \mathsf{Bool}$.

- Construct a map $\neg : \mathsf{Bool} \to \mathsf{Bool}$.
- Note that $\neg$ is an equivalence, because $\neg$ is its own inverse $(\neg(\neg b) = b)$.
- As such, we get a path $p_\neg : \mathsf{Bool} = \mathsf{Bool}$

More on univalence

Let's prove $\mathsf{Bool} = \mathsf{Bool}$.

- Construct a map $\neg : \mathsf{Bool} \to \mathsf{Bool}$.
- Note that $\neg$ is an equivalence, because $\neg$ is its own inverse $(\neg(\neg b) = b)$.
- As such, we get a path $p_\neg : \mathsf{Bool} = \mathsf{Bool}$
- Note that $p_\neg \neq \mathsf{refl}_{\mathsf{Bool}}$!
- We apply idtoeqv:

$$\mathsf{idtoeqv}(p_\neg) = \neg, \qquad \mathsf{idtoeqv}(\mathsf{refl}_{\mathsf{Bool}}) = \mathsf{id}$$

$\neg$ and id are not equal

Function extensionality from univalence axiom

Function extensionality follows from univalence. See Section 4.9 in the HoTT book.

Equality in type formers

- The empty type: $\checkmark$
- The unit type: $\checkmark$
- Product types: $\checkmark$
- Sum types: $\checkmark$
- Function types: **function extensionality**
- Dependent products (for all): **function extensionality**
- Dependent sums (there is): $\checkmark$
- Universes: **univalence**

Computing with univalence

- Recall that we constructed an equivalence $\neg : \mathsf{Bool} \to \mathsf{Bool}$.
- Using univalence, we get a path $p_\neg : \mathsf{Bool} = \mathsf{Bool}$.

Computing with univalence

- Recall that we constructed an equivalence $\neg : \mathsf{Bool} \to \mathsf{Bool}$.
- Using univalence, we get a path $p_\neg : \mathsf{Bool} = \mathsf{Bool}$.
- We can construct a boolean

$$\mathsf{idtoeqv}(\mathsf{ua}(\neg))(\mathsf{true})$$

Computing with univalence

- Recall that we constructed an equivalence $\neg : \mathsf{Bool} \to \mathsf{Bool}$.
- Using univalence, we get a path $p_\neg : \mathsf{Bool} = \mathsf{Bool}$.
- We can construct a boolean

$$\mathsf{idtoeqv}(\mathsf{ua}(\neg))(\mathsf{true})$$

- This term is in normal form! We cannot simplify it further, and we cannot reduce it to either true or false

Upcoming talks

- In the remainder of the talks, we study type theory extended with univalence axiom

Upcoming talks

- In the remainder of the talks, we study type theory extended with univalence axiom
- **Problem**: the univalence axiom is an axiom. It does not have any computation rules. As such, we cannot simplify terms that make use of the univalence axiom.

Upcoming talks

- In the remainder of the talks, we study type theory extended with univalence axiom
- **Problem**: the univalence axiom is an axiom. It does not have any computation rules. As such, we cannot simplify terms that make use of the univalence axiom.
- If we see our terms as programs, this means that we have programs that can get stuck on certain computations, because we don't have rules to simplify them further.

Upcoming talks

- In the remainder of the talks, we study type theory extended with univalence axiom
- **Problem**: the univalence axiom is an axiom. It does not have any computation rules. As such, we cannot simplify terms that make use of the univalence axiom.
- If we see our terms as programs, this means that we have programs that can get stuck on certain computations, because we don't have rules to simplify them further.
- So, we want to change our type theory in such a way that we have univalence with computation rules.