The Coherence Problem in HoTT

Thorsten Altenkirch
FP Away Day 2014

Before the revolution…

- In Intensional Type Theory the equality type

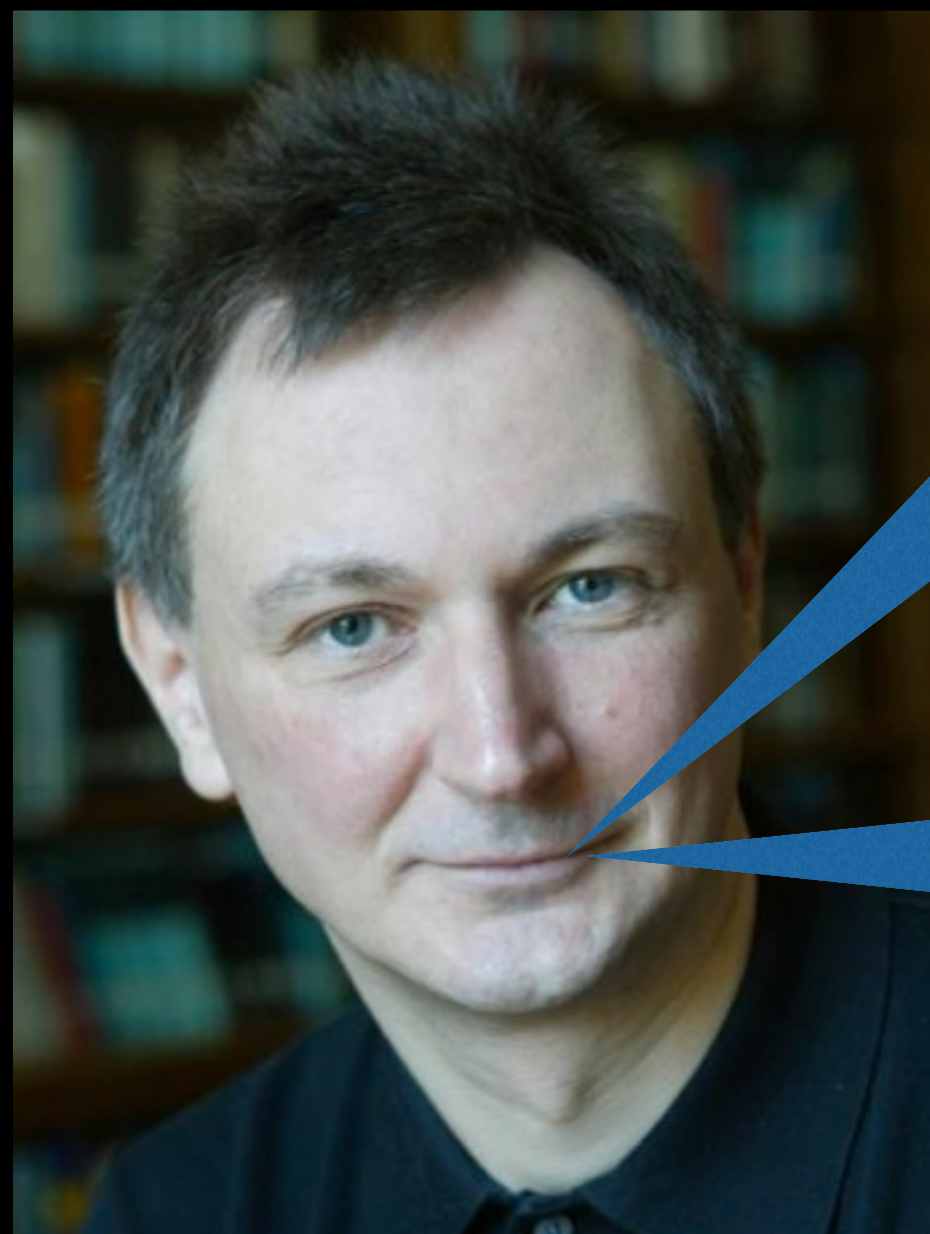
```
data _≡_ {A : Set} (x : A) : A → Set where
             refl : x ≡ x
```

  - reflects definitional equality
  - is proof-irrelevant
  - is not extensional :
    - does not validate functional extensionality
    - does not validate univalent

…after the revolution

- In HoTT the equality type:
  - does not reflect propositional equality
  - is proof relevant
  - is extensional
    - validates functional extensionality
    - validates univalence

Are we happy now?



Univalent equality is what you want to do Mathematics!

But sometimes I would like to have a strict equality …

Vladimir Voevodsky

Voevodsky's exercise

Define Semisimplicial Types in HoTT!

Semisimplicial Types

```
SSType = Σ

  (X₀ : U)

  (X₁ : X₀ → X₀ → U)

  (X₂ : {x₀ x₁ x₂ : X₀})
        → X₁ x₀ x₁ → X₁ x₁ x₂ → X₁ x₀ x₂ → U

  (X₃ : {x₀ x₁ x₂ x₃ : X₀}
        {x₀₁ : X₁ x₀ x₁}{x₁₂ : X₁ x₁ x₂}{x₀₂ : X₁ x₀ x₂}
        {x₀₃ : X₁ x₀ x₃}{x₁₃ : X₁ x₁ x₃}{x₂₃ : X₁ x₂ x₃}
        → X₂ x₀₁ x₁₂ x₀₂
        → X₂ x₀₁ x₁₃ x₀₃
        → X₂ x₀₂ x₂₃ x₀₃
        → X₂ x₁₂ x₂₃ x₁₃
        → U

   ...
```

SSType in old Type Theory

```
record Δ (m n : ℕ) : U where
  field
    f : Fin (suc m) → Fin (suc n)
    isMonotone : monotone f
    isInjective : injective f
```

```
record SSet : U₁ where
  field
    X : ℕ → U
    Xm : ∀ {m}{n} → Δ m n → X n → X m
    Xid : ∀{m}{x : X m} → Xm idΔ x ≡ x
    Xo :  ∀ {l}{m}{n}{f : Δ m n}{g : Δ l m}{x : X n}
            → Xm (f oΔ g) x ≡ Xm g (Xm f x)
```

SSType in HoTT ?

- This does not work in HoTT.
- Equality is proof-relevant!
- Hence to be equivalent to the context we need to add coherence laws.

Coherence Laws?

E.g. There are two ways to prove `Xm (f o∆ id∆) x ≡ Xm f x` :

1. Using `f o∆ id∆ ≡ f` and that Xm preserves equality.
2. Using Xo : `Xm (f o∆ id∆) x ≡ Xm id∆ (Xm f x)`
   and Xid : `Xm id∆ (Xm f x) ≡ Xm f x`
   and transitivity.

And we need them to be equal!

Coherence laws …

- There are infinitely many such laws at higher dimensions.
- Defining the type of coherence laws doesn’t seem easier than defining SSType itself!

Genius needed!

- But maybe there is another way to define SSet avoiding the coherence problem!
- E.g. can't we define the approximations

```
SSetN : ℕ → U
```

  using recursion?
- Many have tried …
- But nobody has succeeded!

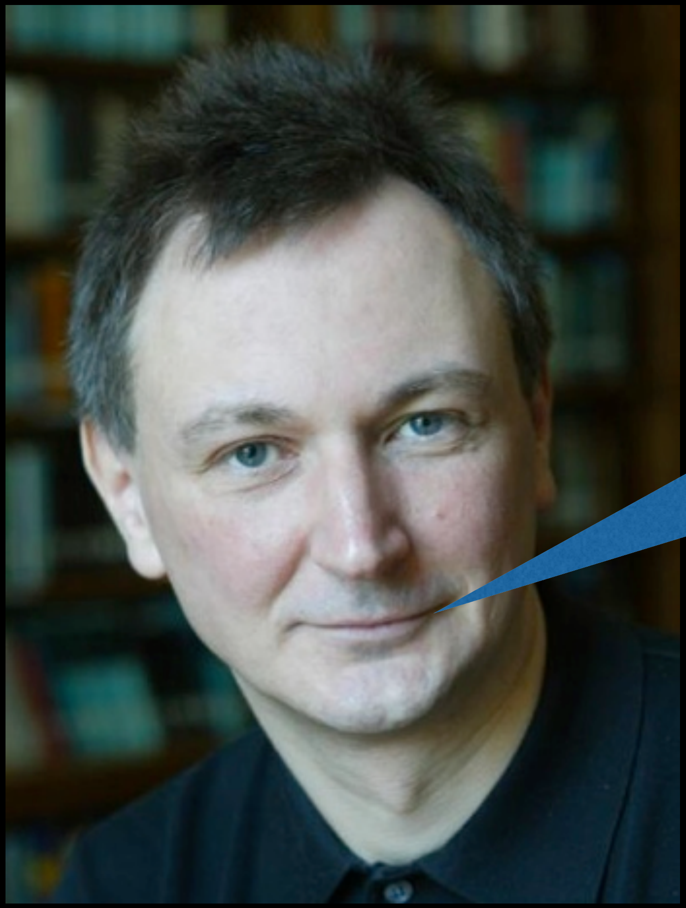
Strict equality?

- But do we really need to solve a coherence problem to define SSType?
- We want the equalities in the presheaf definition to be strict!
- So that the approximations are strictly isomorphic to the corresponding contexts.

Strict Equality ??

- We would like to have access to a strict equality that **reflects** definitional equality.
- Let's write $=$ for the extensional equality from HoTT and $\equiv$ for the strict equality.
- But we cannot have two different equalities because we can show $a = b \;\rightarrow\; a \equiv b$!

HTS ?



I want my own Type Theory where I can have both equalities. I call it HTS (for Homotopy Type System).

I am going to implement it!

Dan Grayson

HTS

- Features extensional equality (=)
  and strict equality (≡)
- Strict equality uses equality reflection (as in NuPRL).
- Hence type-checking is undecidable.
- Distinguishes between pretypes (like $a \equiv b$) and types (like $a = b$).
- Extensional equality can only eliminate over types.
- Hence $a = b$ does not entail $a \equiv b$.

An alternative to HTS

- Since HTS is based on Extensional Type Theory it cannot be easily simulated in Agda or Coq.
- I propose an alternative which I have implemented in Agda
- It also differs from HTS in that we can define dependent types from fibrations.
- I am not yet sure wether I can already define semisimplicial types.

A universe ...

```
data U : Set
El : U → Set

data U where
  UU : U
  π : (A : U)(B : El A → U) → U
  σ : (A : U)(B : El A → U) → U
  Nat : U
  _~_ : {A : U} → El A → El A → U


El UU = U
El (π A B) = (x : El A) → El (B x)
El (σ A B) = Σ (El A) (λ x → El (B x))
El Nat = ℕ
El (a ~ a') = ...
```

… with extensional equality

```
refl~ : {A : U}{x : El A} → El (x ~ x)

J~ : {A : U}{a : El A}
      (P : {a' : El A} → El (a ~ a') → U)
    → El (P refl~)
    → {a' : El A}(p : El (a ~ a')) → El (P p)
```

```
ext : {A : U}{B : El A → U}{f f' : El (π A B)}
    → ({x : El A} → El ((f x) ~ (f' x)))
    → El (f ~ f')
```

```
ua : {A B : El UU}{p : El (A ~ B)} → isEquiv (coe p)
```

- Here Agda's ≡ plays the role of strict equality.
- ~ represents extensional equality.
- Agda types correspond to pretypes.
- While elements of U correspond to proper (extensional) types.
- We cannot prove a ~ b implies a ≡ b because J~ only eliminates over types.

What is a fibration?

```
isFib : {A B : U}(f : El A → El B) → Set
```

```
isFib {A} {B} f = (a : El A)(b : El B)(p : El (f a ~ b))
                → Σ[ a' ∈ El A ]
                  Σ[ q ∈ El (a ~ a') ]
                    _≡_ {A = Σ[ b' ∈ El B ]
                               El (f a ~ b')}
                        (b , p)
                        ((f a') , cong~ f q)
```

Projections are fibrations

```
fst : {A : U}{B : El A → U} → El (σ A B) → El A
fst (a , b) = a
```

```
isFibFst : {A : U}{B : El A → U} → isFib (fst {A} {B})
```

We need an extra assumption:

```
p ≡ cong~ fst cong~[ p , q ]
```

A new type former

```
data U where
  ...
  Fam : {A B : U}(f : El A → El B) → isFib f → El B → U
```

```
El (Fam {A} f _ b) = Σ[ a ∈ El A ] f a ≡ b
```

Conclusions

- We can show that types are closed under strict pullbacks.
- We can also define strict versions of certain presheaf categories.
- Can we define SSType?