An Introduction to Axel Ljungström's PhD thesis:

Synthetic approaches to cohomology, homology and homotopy

Nicolai Kraus

Stockholm, 21 May 2025

Analysing the Thesis Title

Synthetic approaches *to* *cohomology, homology and homotopy*

Done in a setting where:

Primitive things = Things of interest

(i.e., things of interest are native, not defined)

Things of interest = Spaces

Analysing the Thesis Title

Synthetic approaches *to* *cohomology, homology and homotopy*

Done in a setting where:

Primitive things = Things of interest

(i.e., things of interest are native, not defined)

Things of interest = Spaces

Thesis setting/language: Homotopy Type Theory

Dependent Type Theory

Idris

Agda

LEAN THEOREM PROVER

Arend

Rocq
/Coq

Types in Programming

Examples: int, double, bool

useful for catching mistakes, partial documentation:

```
int calculatePrime(int n) {

    ...

}
```

Types in Programming

Examples: int, double, bool

useful for catching mistakes, partial documentation:

```
int calculatePrime(int n) {

  return 7;

}
```

Dependent Types (eg Agda)

```
calculatePrime : (n : ℕ) → Σ[ p : ℕ ]  (isPrime p) × (p > n)
calculatePrime = ?
```

Dependent Types (eg Agda)

```
calculatePrime : (n : ℕ) → Σ[ p : ℕ ]  (isPrime p) × (p > n)
calculatePrime = ?
```

root of syntax tree

Primes and twin primes

Consider two exercises in Agda:

```
calculatePrime : (n : ℕ) → Σ[ p : ℕ ], (isPrime p) × (p > n)
calculatePrime = ?


calcTwinPrime : (n : ℕ) → Σ[ p : ℕ ], (isPrime p) × (p > n) × (isPrime (p + 2))
calcTwinPrime = ?
```

Primes and twin primes

Consider two exercises in Agda:

```
calculatePrime : (n : ℕ) → Σ[ p : ℕ ], (isPrime p) × (p > n)
calculatePrime = ?
```

```
calcTwinPrime : (n : ℕ) → Σ[ p : ℕ ], (isPrime p) × (p > n) × (isPrime (p + 2))
calcTwinPrime = ?
```

Agda type- and termination-checks. **Programming = Proving**

What is a type?

We see:	We might think of:
$\mathbb{N}$	set {0,1,2,...}
`p > n`	a proposition
`isPrime p`	a proposition
`type A`	an unspecified set (?)
`a term x : A`	an element of the set (?)

What is a type?

Syntax (mostly determined by the type theory)

We see:	We might think of:
$\mathbb{N}$	set {0,1,2,...}
`p > n`	a proposition
`isPrime p`	a proposition
`type A`	an unspecified set (?)
`a term x : A`	an element of the set (?)

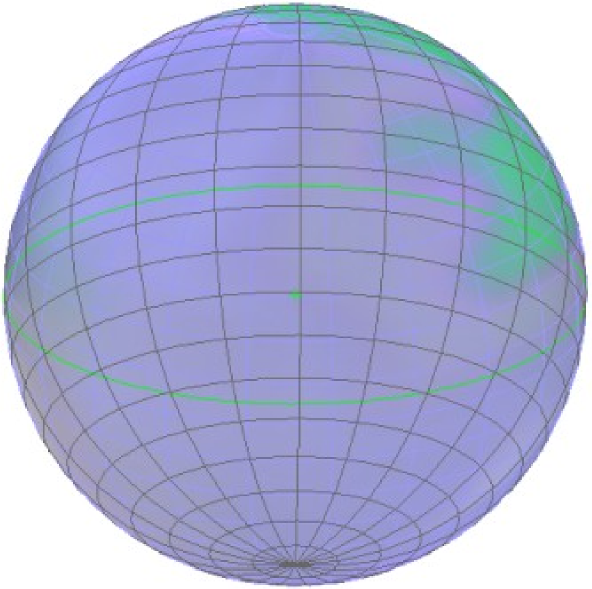
Semantics (our choice!)

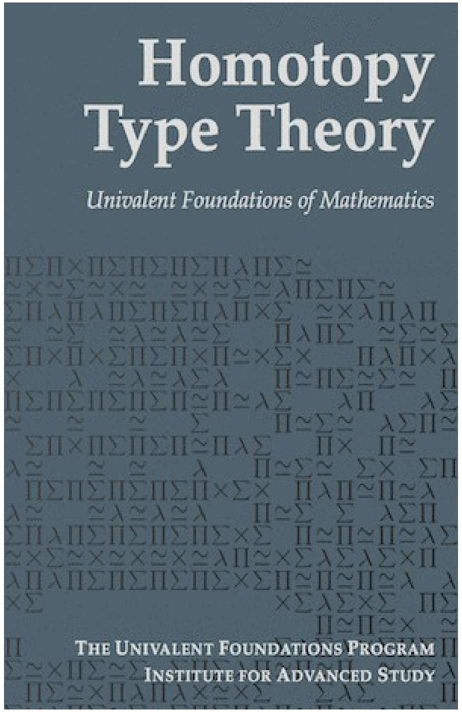
Interpretations

Type Theory	Set Theory	Logic
$A : \mathcal{U}$	A is a set	A is a proposition
$x : A$	$x \in A$	x is a proof of A
$A \to B$	$A \to B$	A implies B
$B : A \to \mathcal{U}$	B_a is a family of sets	B is a predicate
$(a : A) \to B(a)$	$\prod_{a \in A} B_a$	$\forall\, a \,.\, B(a)$
$(a : A) \times B(a)$	$\coprod_{a \in A} B_a$	$\exists\, a \,.\, B(a)$
$x = y$	$x = y$	$x = y$

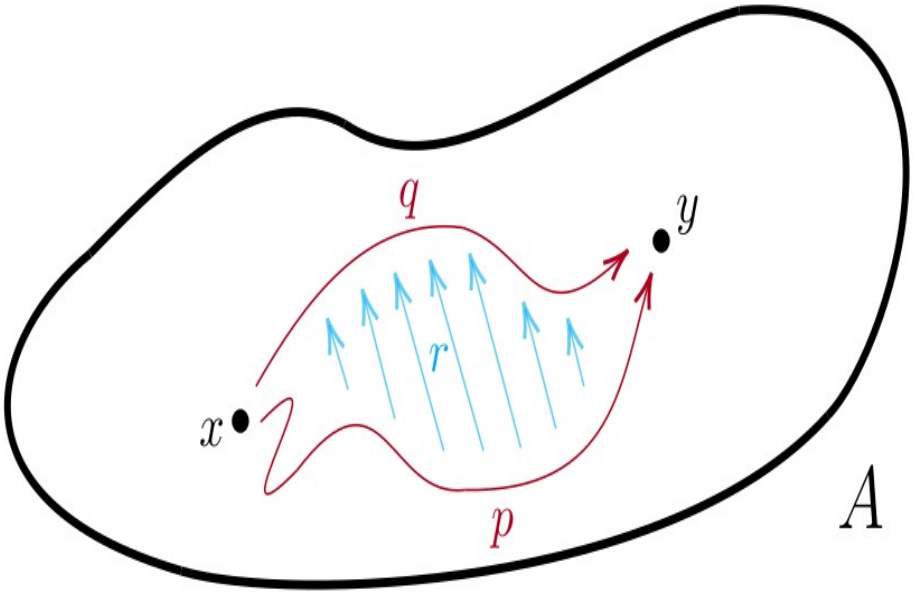
(table 1 from Axel's thesis)

HoTT: view types as spaces





Interpreting types as spaces



Type theory	Homotopy theory
$A : \mathcal{U}$	A is a space
$x : A$	x is a point in A
$A \to B$	$A \to B$ (cont.)
$B : A \to \mathcal{U}$	B is a fibration
$(a : A) \to B(a)$	Sections of B
$(a : A) \times B(a)$	Total space of B
$x = y$	Path space $P(x, y)$

(from Axel's thesis)

Martin-Löf's Identity Type

Given a type A and two terms `x, y : A`,
there is a type `(x = y)`.

formation rule

We always have `refl : x = x`.

introduction rule

To define

```
F : (x y : A) → (p : x = y) → C(x,y,p)
```

it suffices to define

```
f': (x : A) → C(x, x, refl).
```

elimination rule
("J")

Examples with =

Exercise:

```
trans : (x y z : A) →
        (x = y) → (y = z) → (x = z)
```

Solution:

Using the elimination rule for =, we only need

```
trans' : (x z : A) → (x = z) → (x = z)
```

which is easy. 🙂

Examples with =

Exercise:

```
K : (x : A) → (p : x = x) → (p = refl)
```

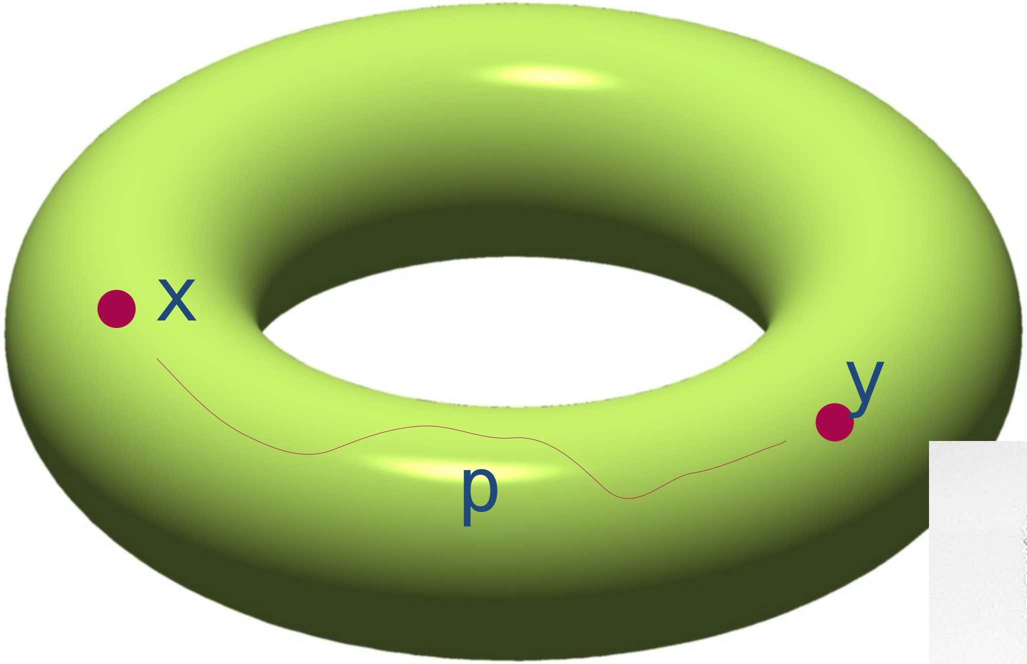
No solution, as shown
by Hofmann and Streicher's
Groupoid Model.

Intuition for =

`A   type`

`x,y : A`

`p : x = y`

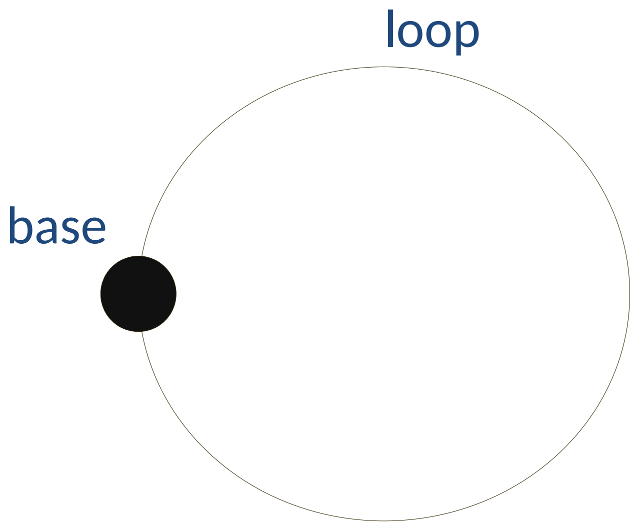


"Types behave like higher groupoids / homotopy types."

Application of "Types as Spaces"

We can define so-called *Higher Inductive Types*.
Basic example:

```
data 𝕊¹ : Type where
  base : 𝕊¹
  loop : base == base
```



Why work in HoTT?

🙂 HoTT is elegant; arguments are reduced to their “mathematical core.”

🙂 HoTT allows computers to check correctness; Axel’s results are mechanized in Cubical Agda.

🙂 Once implemented, results can be calculated by a computer.

🙂 Work in HoTT is very general; we have a model in every ∞-topos.

🙂 The language only allows “good” constructions; avoids ill-defined concepts.

😐 The language only allows “good” constructions; can be very restrictive.

Axel’s PhD Work

I want to emphasize two main points:

1. Axel’s work contains several highly complex results on (co)homology and homotopy theory in HoTT. This is an excellent contribution to the field.

2. Axel’s work is mechanized in Cubical Agda. This gives confidence, computations, and important insights into this (fairly new!) variation of Agda.

Thanks for your attention! (We’re far from done.)