Programs as Proofs:

Why mathematicians become programmers, and computer scientists do homotopy theory

Nicolai Kraus
At Freie Universität Berlin,
2025

University of Nottingham

Dependent Type Theory

Idris

Agda

LEAN THEOREM PROVER

Arend

Rocq
/Coq

Types

Examples: int, double, bool

useful for catching mistakes, partial documentation:

```
int calculatePrime(int n) {

    ...

}
```

Types

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**

Constructive Mathematics

Constructive Mathematics

Definition: A foundation is *constructive* if, whenever we prove that a solution exists, we can calculate it.

(… and we can’t do this if we use LEM or AC!)

Same definition in CS language: A proof assistant implements a constructive foundation if we can run our programs.

Constructivity – why care?

1. Philosophical reasons (... no idea)

2. Certain constructive type theories are the internal languages of certain toposes / mathematical objects.

3. In practice: terms reduce; sometimes much easier to work with in proof assistants; we get solutions “for free”.

What is a type?

We see:	We 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 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!)

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:

$$\mathtt{sym} : (x\ y : A) \to \overbrace{(x = y)}^{(p\,:\,x = y)} \to \overbrace{(y = x)}^{C\,(x,y,p)}$$

Solution:

Using the elimination rule for =, we only need

```
sym' : (x : A) → (x = x)
```

which is easy. 🙂

Examples with =

Exercise:

This is $C(x, y, p)$

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

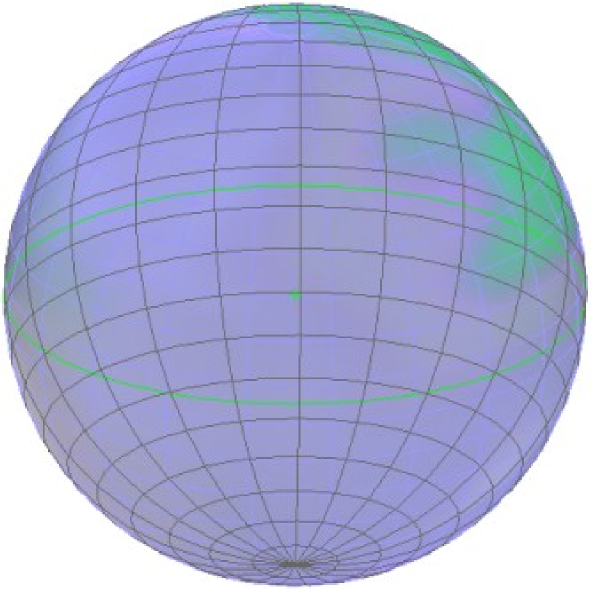
Solution:

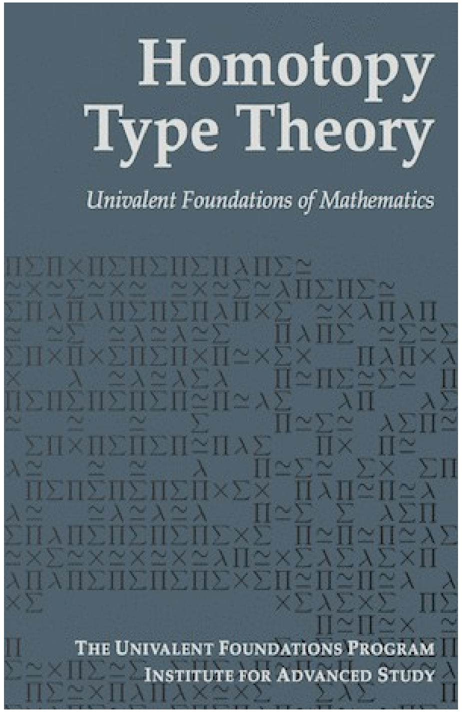
Using the elimination rule for =, we only need

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

which is easy. 🙂

HoTT: view types as spaces





Examples with =

Exercise:

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

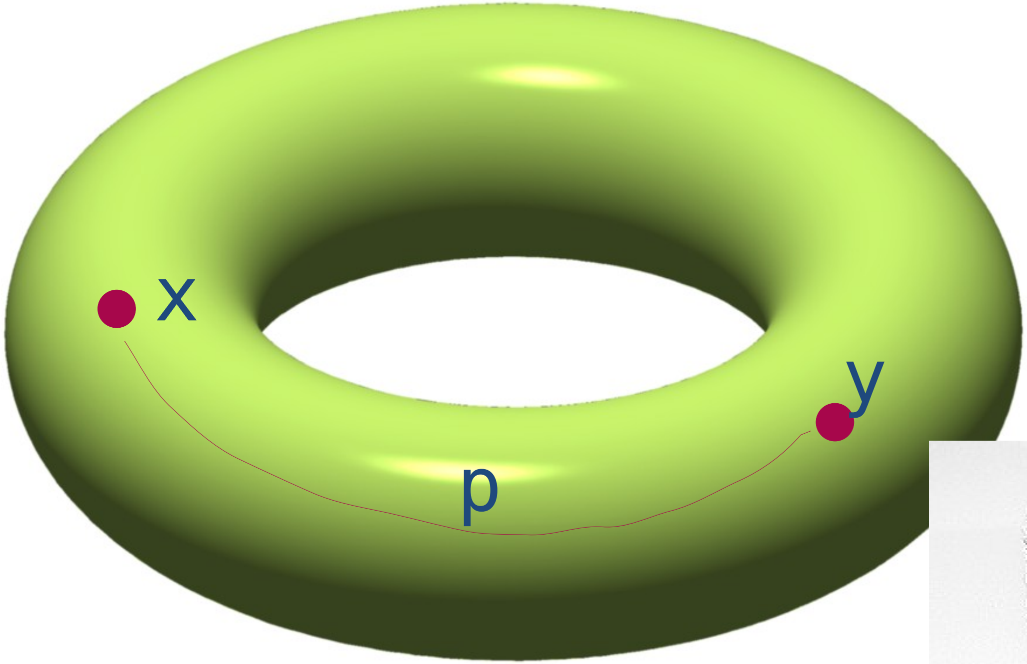
No solution, as shown
by Hofmann and Streicher’s
Groupoid Model.

Intuition for =

A type

x, y : A

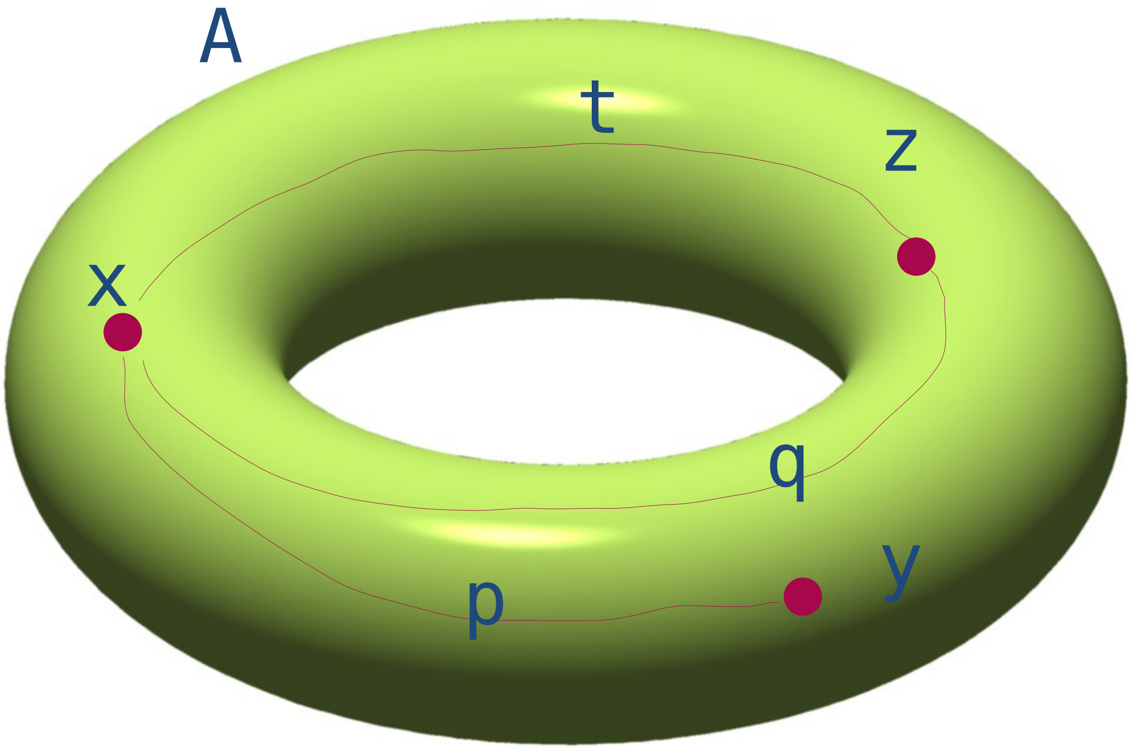
p : x = y



“Types behave like higher groupoids / homotopy types.”

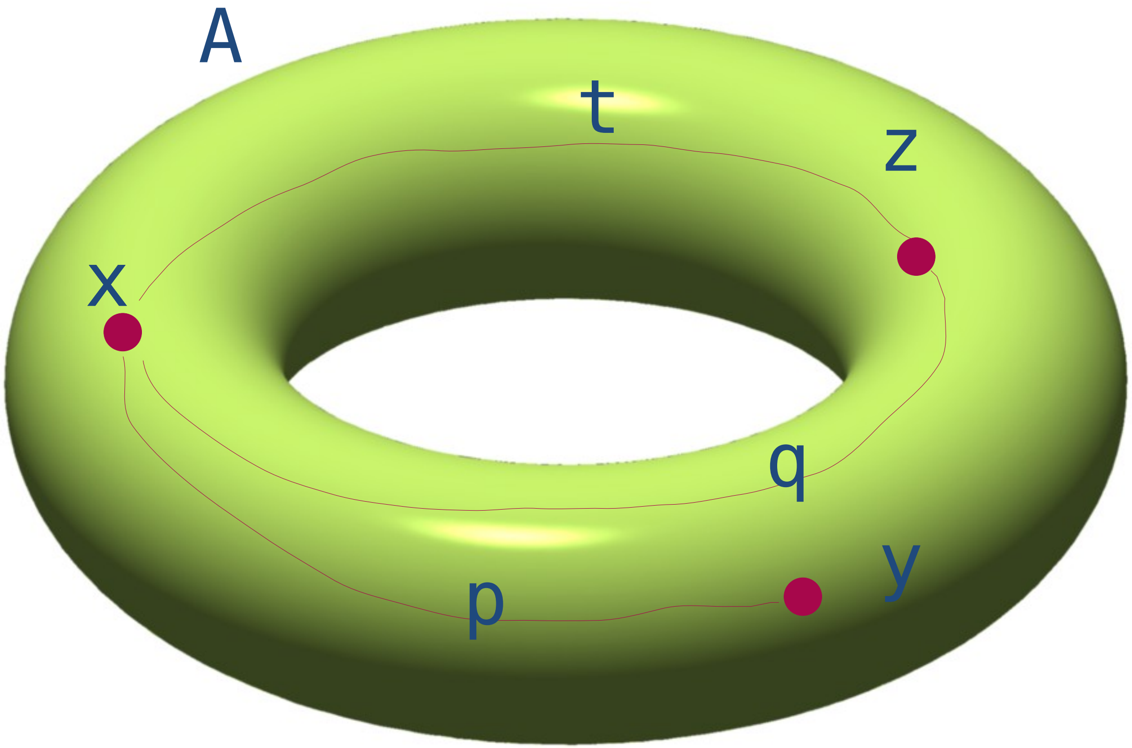
Intuition for =

```
A    type
x,y,z : A
p : x = y
q : x = z
t : x = z
```



Intuition for =

```
A    type
x,y,z : A
p : x = y
q : x = z
t : x = z
```

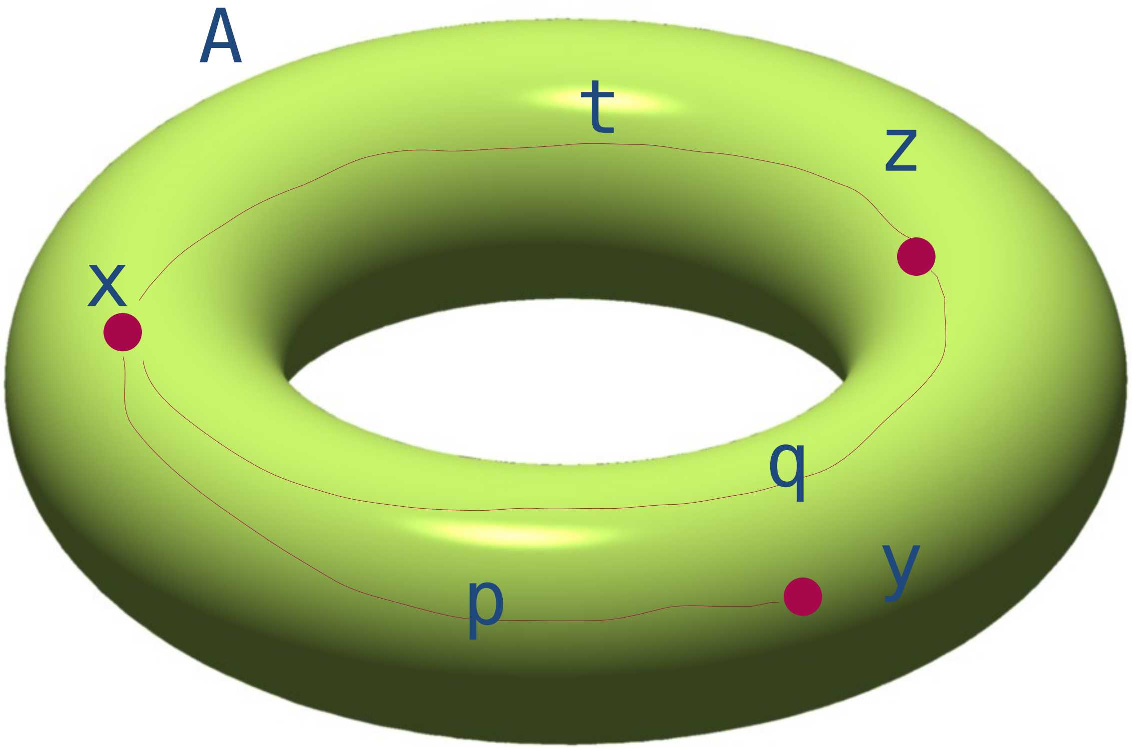


Which of the following make sense and are provable?

1. p == q
2. (y,p) == (z,q)
3. q == t

Intuition for =

A type
x,y,z : A
p : x = y
q : x = z
t : x = z



Which of the following make sense and are provable?

1. p == q
   type error!
2. (y,p) == (z,q)
3. q == t

Intuition for =

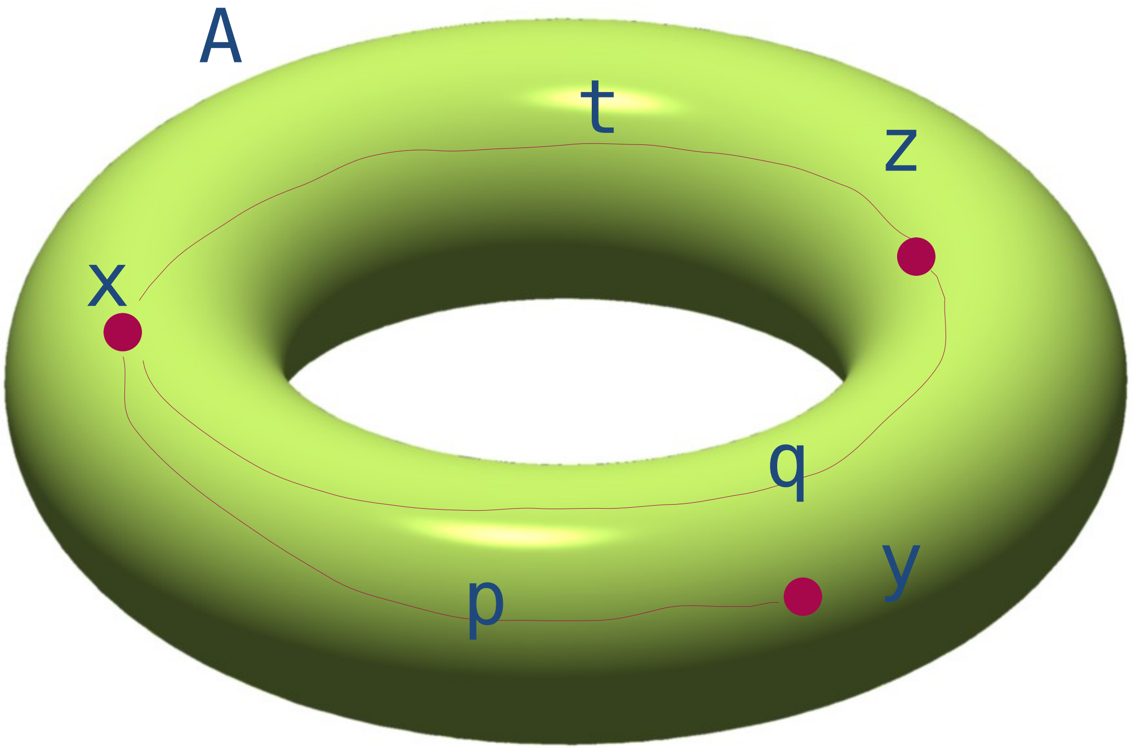
A   type

x,y,z : A

p : x = y

q : x = z

t : x = z



Which of the following make sense and are provable?

1. p == q

   type error!

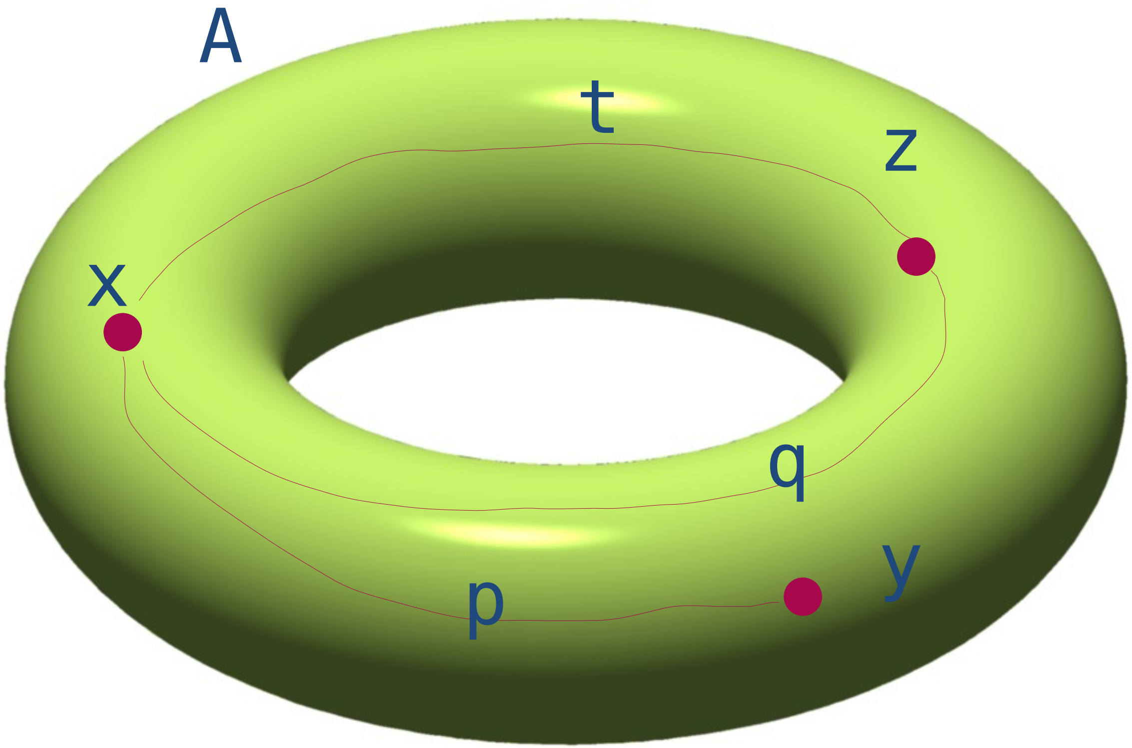
2. (y,p) == (z,q)

   in type: $\Sigma(u{:}A).\ x=u$
   yes!

3. q == t

Intuition for =

```
A    type
x,y,z : A
p : x = y
q : x = z
t : x = z
```

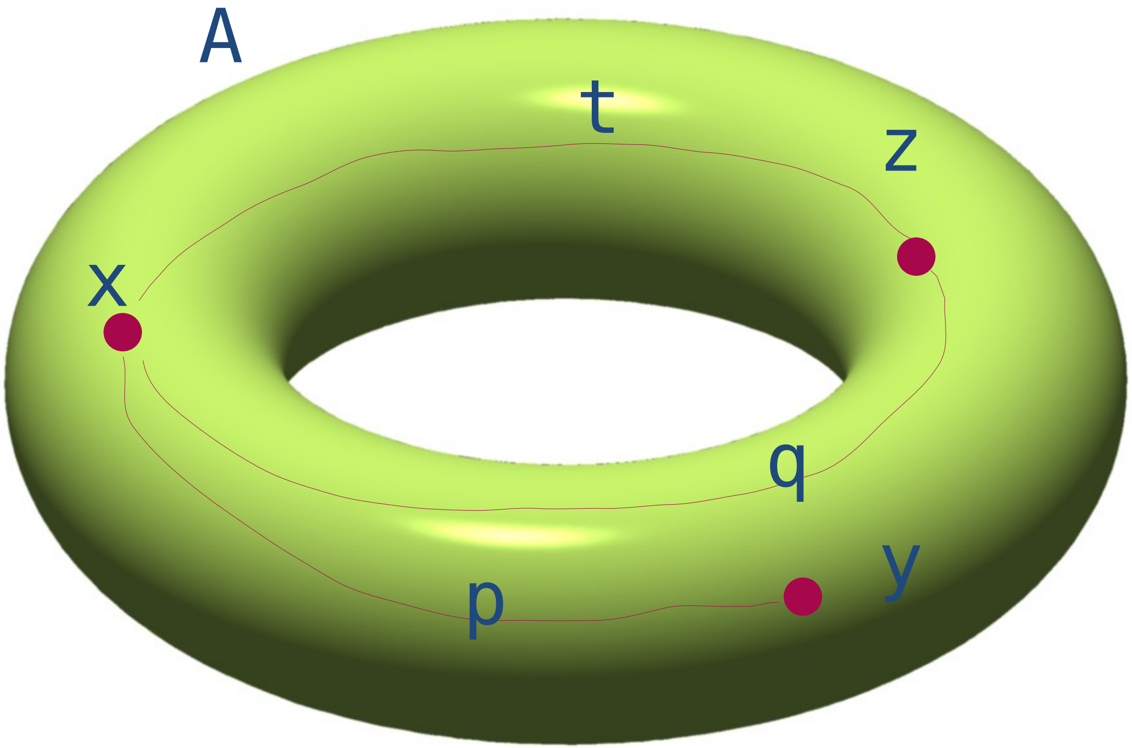


Which of the following make sense and are provable?

1. p == q
   type error!
2. (y,p) == (z,q)
   in type: $\Sigma(u{:}A).\ x = u$
   yes!
3. q == t
   type-checks, but not provable

Intuition for =

```
A    type
x,y,z : A
p : x = y
q : x = z
t : x = z
```



Def (Voevodsky):
A type X is *contractible* if
$\Sigma(x_0 : X).$
$(y : X) \to x_0 = y$
is inhabited.

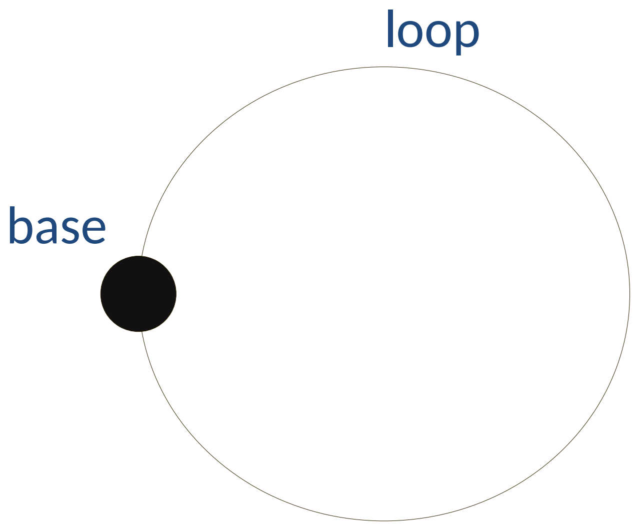
Question:
Is the torus (A) contractible?

Application 1: Circle

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

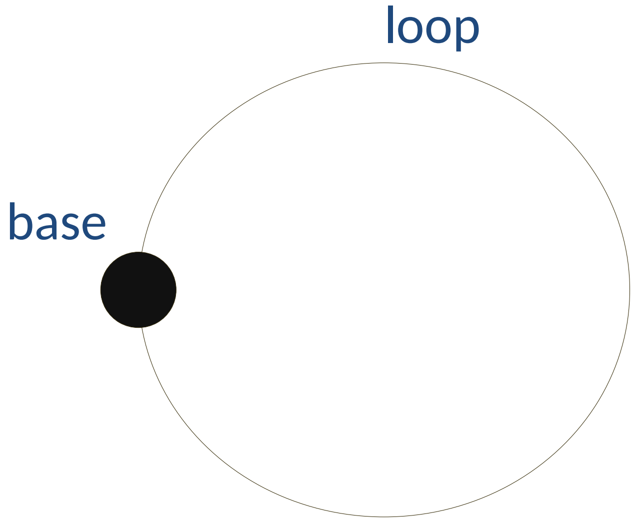
Application 1: Circle

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



Application 1: Circle

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



"Synthetic homotopy theory"

Example result: $\pi_4(S^3) \simeq \mathbb{Z}/2\mathbb{Z}$ (Brunerie)

Application 2: Groups

```
record aGroup : Type₁ where
  field

    G : Set
    _·_ : G → G → G
    assoc : ∀{x y z} → ((x · y) · z) == (x · (y · z))

    e : G
    e-right : ∀{x} → (x · e) == x
    e-left  : ∀{x} → (e · x) == x

    inv : G → G
    inv-left :  ∀{x} → (inv x · x) == e
    inv-right : ∀{x} → (x · inv x) == e
```

Application 2: Groups

```
record aGroup : Type₁ where
  field

    G : Set
    _·_ : G → G → G
    assoc : ∀{x y z} → ((x · y) · z) == (x · (y · z))

    e : G
    e-right : ∀{x} → (x · e) == x
    e-left  : ∀{x} → (e · x) == x

    inv : G → G
    inv-left :  ∀{x} → (inv x · x) == e
    inv-right : ∀{x} → (x · inv x) == e
```

```
record cGroup : Type₁ where
  field

    X : Type
    x : X
    h : is-1-type X
    c : is-connected X
```

Application 2: Groups

```
record aGroup : Type₁ where
  field

    G : Set
    _·_ : G → G → G
    assoc : ∀{x y z} → ((x · y) · z) == (x · (y · z))

    e : G
    e-right : ∀{x} → (x · e) == x
    e-left  : ∀{x} → (e · x) == x

    inv : G → G
    inv-left :  ∀{x} → (inv x · x) == e
    inv-right : ∀{x} → (x · inv x) == e
```

```
record cGroup : Type₁ where
  field

    X : Type
    x : X
    h : is-1-type X
    c : is-connected X
```

Given a concrete Group (X,x,h,c),
we can construct an abstract group by setting:

G := (x==x)

e := refl

inv := sym (and so on)

“Mathematical DSLs”

Martin-Löf type theory
(mechanization of maths, verified programming)

Directed type theories
(for directed higher structures)

Homotopy Type Theory
(same as MLTT,
plus synthetic homotopy theory)

Two-level type theory
(framework for extensions, study meta-theory)

Modal type theory
(if modalities are needed)

Cubical Type Theory
(better computation, but fewer models than HoTT)

(and so on)

“Mathematical DSLs”

Martin-Löf type theory
(mechanization of maths,
verified programming)

Directed type theories
(for directed higher
structures)

Homotopy Type Theory
(same as MLTT,
plus synthetic homotopy theor

Thanks!

Two-level type
theory
(framework for
extensions, study
meta-theory)

Modal type theory
(if modalities are
needed)

Cubical Type Theory
(better computation,
but fewer models than
HoTT)

(and so on)