CSC_51051_EP: Homotopy types

Samuel Mimram

2024

École polytechnique

Part I

Equality

Identity types

Recall that in Agda, we have to notions of equality:

Identity types

Recall that in Agda, we have to notions of equality:

- **definitional equality**: we cannot distinguish between $\alpha\beta\eta$-equivalent terms

Identity types

Recall that in Agda, we have to notions of equality:

- **definitional equality**: we cannot distinguish between $\alpha\beta\eta$-equivalent terms
- **propositional equality**: the ≡ predicate that we defined.

Identity types

Recall that in Agda, we have to notions of equality:

- **definitional equality**: we cannot distinguish between $\alpha\beta\eta$-equivalent terms
- **propositional equality**: the $\equiv$ predicate that we defined.

We call $t \equiv u$ an **identity type** and sometimes write it $\mathsf{Id}_A(t, u)$.

Identity types

Recall that in Agda, we have to notions of equality:

- **definitional equality**: we cannot distinguish between $\alpha\beta\eta$-equivalent terms
- **propositional equality**: the $\equiv$ predicate that we defined.

We call $t \equiv u$ an **identity type** and sometimes write it $\mathsf{Id}_A(t, u)$.

In case you forgot,

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

Identity types

Recall that in Agda, we have to notions of equality:

- **definitional equality**: we cannot distinguish between $\alpha\beta\eta$-equivalent terms
- **propositional equality**: the $\equiv$ predicate that we defined.

We call $t \equiv u$ an **identity type** and sometimes write it $\mathrm{Id}_A(t, u)$.

In case you forgot,

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

It is of course possible to directly give the rules satisfied by those types.

Identity types

Note that definitional equality implies propositional equality: the rule

$$\frac{\Gamma \vdash t = u : A}{\Gamma \vdash \mathsf{refl} : \mathsf{Id}_A(t, u)}$$

is admissible.

Rules for identity types

Formation:

Rules for identity types

Formation:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Id}_A(t, u) : \mathsf{Type}}\ (\mathsf{Id_F})$$

Rules for identity types

Formation:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Id}_A(t, u) : \mathsf{Type}} \; (\mathsf{Id_F})$$

Introduction:

Rules for identity types

Formation:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : A}{\Gamma \vdash \mathsf{Id}_A(t, u) : \mathsf{Type}} \; (\mathsf{Id}_\mathsf{F})$$

Introduction:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathsf{refl}(t) : \mathsf{Id}_A(t, t)} \; (\mathsf{Id}_\mathsf{I})$$

Rules for identity types

Elimination:

Rules for identity types

Elimination:

$$\dfrac{\Gamma \vdash p : \mathsf{Id}_A(t, u) \qquad \Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash B : \mathsf{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathsf{refl}(x)/z]}{\Gamma \vdash \mathsf{J}(p, xyz \mapsto B, x \mapsto r) : B[t/x, u/y, p/z]}\ (\mathsf{Id_E})$$

In Agda:

```
J : {A : Set} (B : (x y : A) → x ≡ y → Set) (r : (x : A) → B x x refl)
    (t : A) (u : A) (p : t ≡ u) → B t u p
J B r t .t refl = r t
```

Rules for identity types

Elimination:

$$\frac{\Gamma \vdash p : \mathrm{Id}_A(t, u) \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathrm{refl}(x)/z]}{\Gamma \vdash \mathrm{J}(p, xyz \mapsto B, x \mapsto r) : B[t/x, u/y, p/z]} \; (\mathrm{Id_E})$$

Computation:

In Agda:

```
J : {A : Set} (B : (x y : A) → x ≡ y → Set) (r : (x : A) → B x x refl)
    (t : A) (u : A) (p : t ≡ u) → B t u p
J B r t .t refl = r t
```

Rules for identity types

Elimination:

$$\frac{\Gamma \vdash p : \mathsf{Id}_A(t, u) \qquad \Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash B : \mathsf{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathsf{refl}(x)/z]}{\Gamma \vdash \mathsf{J}(p, xyz \mapsto B, x \mapsto r) : B[t/x, u/y, p/z]} \; (\mathsf{Id_E})$$

Computation:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash B : \mathsf{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathsf{refl}(x)/z]}{\Gamma \vdash \mathsf{J}(\mathsf{refl}(t), xyz \mapsto B, x \mapsto r) = r[t/x] : B[t/x, t/y, \mathsf{refl}(t)/z]} \; (\mathsf{Id_C})$$

In Agda:

```
J : {A : Set} (B : (x y : A) → x ≡ y → Set) (r : (x : A) → B x x refl)
    (t : A) (u : A) (p : t ≡ u) → B t u p
J B r t .t refl = r t
```

Rules for identity types

Elimination:

$$\frac{\Gamma \vdash p : \mathrm{Id}_A(t,u) \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x,y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathrm{refl}(x)/z]}{\Gamma \vdash \mathrm{J}(p, xyz \mapsto B, x \mapsto r) : B[t/x, u/y, p/z]} \ (\mathrm{Id_E})$$

Computation:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x,y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathrm{refl}(x)/z]}{\Gamma \vdash \mathrm{J}(\mathrm{refl}(t), xyz \mapsto B, x \mapsto r) = r[t/x] : B[t/x, t/y, \mathrm{refl}(t)/z]} \ (\mathrm{Id_C})$$

Uniqueness:

Rules for identity types

Elimination:

$$\frac{\Gamma \vdash p : \mathrm{Id}_A(t, u) \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathrm{refl}(x)/z]}{\Gamma \vdash \mathrm{J}(p, xyz \mapsto B, x \mapsto r) : B[t/x, u/y, p/z]} \ (\mathrm{Id_E})$$

Computation:

$$\frac{\Gamma \vdash t : A \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A \vdash r : B[x/x, x/y, \mathrm{refl}(x)/z]}{\Gamma \vdash \mathrm{J}(\mathrm{refl}(t), xyz \mapsto B, x \mapsto r) = r[t/x] : B[t/x, t/y, \mathrm{refl}(t)/z]} \ (\mathrm{Id_C})$$

Uniqueness:

$$\frac{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash t : B}{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash \mathrm{J}(z, xyz \mapsto B, x \mapsto t[x/y, \mathrm{refl}(x)/z]) = t : B} \ (\mathrm{Id_U})$$

About the uniqueness rule

The uniqueness rule is problematic:

$$\frac{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash B : \mathrm{Type} \qquad \Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash t : B}{\Gamma, x : A, y : A, z : \mathrm{Id}_A(x, y) \vdash \mathrm{J}(z, xyz \mapsto B, x \mapsto t[x/y, \mathrm{refl}(x)/z]) = t : B} \; (\mathrm{Id}_{\mathrm{U}})$$

About the uniqueness rule

The uniqueness rule is problematic:

$$\frac{\Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash B : \mathsf{Type} \qquad \Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash t : B}{\Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash \mathsf{J}(z, xyz \mapsto B, x \mapsto t[x/y, \mathsf{refl}(x)/z]) = t : B} \ (\mathsf{Id_U})$$

Namely, one can show that it implies the admissibility of *equality reflection*:

$$\frac{\Gamma \vdash p : \mathsf{Id}_A(t, u)}{\Gamma \vdash t = u : A}$$

About the uniqueness rule

The uniqueness rule is problematic:

$$\frac{\Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash B : \mathsf{Type} \qquad \Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash t : B}{\Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash \mathsf{J}(z, xyz \mapsto B, x \mapsto t[x/y, \mathsf{refl}(x)/z]) = t : B}\ (\mathsf{Id}_\mathsf{U})$$

Namely, one can show that it implies the admissibility of *equality reflection*:

$$\frac{\Gamma \vdash p : \mathsf{Id}_A(t, u)}{\Gamma \vdash t = u : A}$$

which makes typechecking undecidable...

About the uniqueness rule

The uniqueness rule is problematic:

$$\frac{\Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash B : \mathsf{Type} \qquad \Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash t : B}{\Gamma, x : A, y : A, z : \mathsf{Id}_A(x, y) \vdash \mathsf{J}(z, xyz \mapsto B, x \mapsto t[x/y, \mathsf{refl}(x)/z]) = t : B} \ (\mathsf{Id}_\mathsf{U})$$

Namely, one can show that it implies the admissibility of *equality reflection*:

$$\frac{\Gamma \vdash p : \mathsf{Id}_A(t, u)}{\Gamma \vdash t = u : A}$$

which makes typechecking undecidable...

For this reason, it is usually not taken in account.

Leibniz equality

The definition of equality proposed by Leibniz was that two things should be considered as equal when they cannot be distinguished.

Leibniz equality

The definition of equality proposed by Leibniz was that two things should be considered as equal when they cannot be distinguished.

Formally, t and u of type A cannot be distinguished when

Leibniz equality

The definition of equality proposed by Leibniz was that two things should be considered as equal when they cannot be distinguished.

Formally, t and u of type A cannot be distinguished when for every predicate $P : A \to \mathsf{Type}$, if $P\,t$ holds then $P\,u$ holds.

Leibniz equality

The definition of equality proposed by Leibniz was that two things should be considered as equal when they cannot be distinguished.

Formally, t and u of type A cannot be distinguished when for every predicate $P : A \to \mathsf{Type}$, if $P\,t$ holds then $P\,u$ holds.

In Agda, we can define the Leibniz equality by

Leibniz equality

The definition of equality proposed by Leibniz was that two things should be considered as equal when they cannot be distinguished.

Formally, t and u of type A cannot be distinguished when for every predicate $P : A \to \mathrm{Type}$, if $P\,t$ holds then $P\,u$ holds.

In Agda, we can define the Leibniz equality by

```
_≡₁_ : {A : Set} (x y : A) → Set₁
_≡₁_ {A} x y = (P : A → Set) → P x → P y
```

Leibniz equality

The definition of equality proposed by Leibniz was that two things should be considered as equal when they cannot be distinguished.

Formally, t and u of type A cannot be distinguished when for every predicate $P : A \to \mathrm{Type}$, if $P\,t$ holds then $P\,u$ holds.

In Agda, we can define the Leibniz equality by

```
_≡₁_ : {A : Set} (x y : A) → Set₁
_≡₁_ {A} x y = (P : A → Set) → P x → P y
```

It is not clear at all that this is an equivalence relation, but we will see that it is the case.

Leibniz equality

Two equal terms are undistinguishible:

Leibniz equality

Two equal terms are undistinguishible:

```
leibniz : {A : Set} {x y : A} → x ≡ y → (P : A → Set) → P x → P y
leibniz refl P p = p
```

Leibniz equality

Two equal terms are undistinguishible:

```
leibniz : {A : Set} {x y : A} → x ≡ y → (P : A → Set) → P x → P y
leibniz refl P p = p
```

and two undistinguishible terms are equal:

Leibniz equality

Two equal terms are undistinguishible:

```
leibniz : {A : Set} {x y : A} → x ≡ y → (P : A → Set) → P x → P y
leibniz refl P p = p
```

and two undistinguishible terms are equal:

```
leibniz' : {A : Set} {x y : A} → ((P : A → Set) → P x → P y) → x ≡ y
leibniz' {x = x} F = F (λ y → x ≡ y) refl
```

Leibniz equality

Since equality $\equiv$ is an equivalence relation, Leibniz equality also is.

Leibniz equality

Since equality $\equiv$ is an equivalence relation, Leibniz equality also is.

This could also have been shown more directly, e.g.

```
leibniz-refl : {A : Set} {x : A} → ((P : A → Set) → P x → P x)
leibniz-refl {x = x} P p = p
```

Leibniz equality

Since equality $\equiv$ is an equivalence relation, Leibniz equality also is.

This could also have been shown more directly, e.g.

```
leibniz-refl : {A : Set} {x : A} → ((P : A → Set) → P x → P x)
leibniz-refl {x = x} P p = p
```

and

```
leibniz-sym  : {A : Set} {x y : A} →
               ((P : A → Set) → P x → P y) →
               ((P : A → Set) → P y → P x)
leibniz-sym {x = x} F P = F (λ y → (P y → P x)) (λ p → p)
```

Part II

The axioms K and UIP

Are proofs of identity unique?

An interesting question about identity types is:

are two proofs of $t \equiv u$ necessarily the same?

Are proofs of identity unique?

An interesting question about identity types is:

are two proofs of $t \equiv u$ necessarily the same?

in particular, is `refl` *the only proof of $t \equiv t$?*

Are proofs of identity unique?

An interesting question about identity types is:

are two proofs of $t \equiv u$ *necessarily the same?*

in particular, is `refl` *the only proof of* $t \equiv t$*?*

In order to study this, we can formulate the two following axioms:

Are proofs of identity unique?

An interesting question about identity types is:

are two proofs of $t \equiv u$ necessarily the same?

in particular, is `refl` *the only proof of $t \equiv t$?*

In order to study this, we can formulate the two following axioms:

- UIP: uniqueness of identity proofs

```
UIP : Set₁
UIP = {A : Set} {x y : A} (p q : x ≡ y) → p ≡ q
```

Are proofs of identity unique?

An interesting question about identity types is:

are two proofs of $t \equiv u$ necessarily the same?

in particular, is `refl` *the only proof of $t \equiv t$?*

In order to study this, we can formulate the two following axioms:

- UIP: uniqueness of identity proofs

```
UIP : Set₁
UIP = {A : Set} {x y : A} (p q : x ≡ y) → p ≡ q
```

- K:

```
K : Set₁
K = {A : Set} {x : A} (P : (x ≡ x) → Set) →
    P refl → (p : x ≡ x) → P p
```

UIP vs K

Both axioms are equivalent:

```
UIP-K : UIP → K
UIP-K UIP P Pr p = subst P (UIP refl p) Pr
```

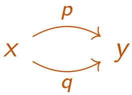
UIP vs K

Both axioms are equivalent:

```
UIP-K : UIP → K
UIP-K UIP P Pr p = subst P (UIP refl p) Pr
```

and

```
loop-eq : {A : Set} {x y : A} (p q : x ≡ y) →
          trans (sym p) q ≡ refl → p ≡ q
loop-eq refl q h = sym h
```



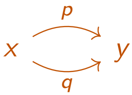
UIP vs K

Both axioms are equivalent:

```
UIP-K : UIP → K
UIP-K UIP P Pr p = subst P (UIP refl p) Pr
```

and

```
loop-eq : {A : Set} {x y : A} (p q : x ≡ y) →
          trans (sym p) q ≡ refl → p ≡ q
loop-eq refl q h = sym h
```




```
K-UIP : K → UIP
K-UIP K p q = loop-eq p q (K (λ r → r ≡ refl) refl (trans (sym p) q))
```

Proving UIP

It turns out that by the usual proof technique, we can prove UIP:

```
UIP-proof : {A : Set} {x y : A} (p q : x ≡ y) → p ≡ q
UIP-proof refl refl = refl
```

Proving UIP

It turns out that by the usual proof technique, we can prove UIP:

```
UIP-proof : {A : Set} {x y : A} (p q : x ≡ y) → p ≡ q
UIP-proof refl refl = refl
```

and K:

```
K-proof : {A : Set} {x : A} (P : (x ≡ x) → Set) →
          P refl → (p : x ≡ x) → P p
K-proof P Pr refl = Pr
```

Proving UIP

It turns out that by the usual proof technique, we can prove UIP:

```
UIP-proof : {A : Set} {x y : A} (p q : x ≡ y) → p ≡ q
UIP-proof refl refl = refl
```

and K:

```
K-proof : {A : Set} {x : A} (P : (x ≡ x) → Set) →
          P refl → (p : x ≡ x) → P p
K-proof P Pr refl = Pr
```

So, case settled?

Part III

Types as spaces

With or without K?

Surprisingly, K cannot be proved using J only (without pattern matching):

```
J : {A : Set} (B : (x y : A) → x ≡ y → Set) (r : (x : A) → B x x refl)
    (t : A) (u : A) (p : t ≡ u) → B t u p
J B r t .t refl = r t
```

With or without K?

Surprisingly, K cannot be proved using J only (without pattern matching):

```
J : {A : Set} (B : (x y : A) → x ≡ y → Set) (r : (x : A) → B x x refl)
    (t : A) (u : A) (p : t ≡ u) → B t u p
J B r t .t refl = r t
```

If we try to translate the proof

```
K-proof : {A : Set} {x : A} (P : (x ≡ x) → Set) →
          P refl → (p : x ≡ x) → P p
K-proof P Pr refl = Pr
```

With or without K?

Surprisingly, K cannot be proved using J only (without pattern matching):

```
J : {A : Set} (B : (x y : A) → x ≡ y → Set) (r : (x : A) → B x x refl)
    (t : A) (u : A) (p : t ≡ u) → B t u p
J B r t .t refl = r t
```

If we try to translate the proof

```
K-proof : {A : Set} {x : A} (P : (x ≡ x) → Set) →
          P refl → (p : x ≡ x) → P p
K-proof P Pr refl = Pr
```

we begin with something like

```
K : {A : Set} {x : A} (P : (x ≡ x) → Set) → P refl → (p : x ≡ x) → P p
K P Pr p = J (λ x y p → P p) ? ? ? ?
```

With or without K?

Surprisingly, K cannot be proved using J only (without pattern matching):

```
J : {A : Set} (B : (x y : A) → x ≡ y → Set) (r : (x : A) → B x x refl)
    (t : A) (u : A) (p : t ≡ u) → B t u p
J B r t .t refl = r t
```

If we try to translate the proof

```
K-proof : {A : Set} {x : A} (P : (x ≡ x) → Set) →
          P refl → (p : x ≡ x) → P p
K-proof P Pr refl = Pr
```

we begin with something like

```
K : {A : Set} {x : A} (P : (x ≡ x) → Set) → P refl → (p : x ≡ x) → P p
K P Pr p = J (λ x y p → P p) ? ? ? ?
```

but this does not type because `p` is of type `x ≡ y`.

With or without K?

It turns out that there are models of dependent type theory in which UIP/K is not validated.

In fact, the default pattern matching algorithm of Agda is too liberal!

With or without K?

It turns out that there are models of dependent type theory in which UIP/K is not validated.

In fact, the default pattern matching algorithm of Agda is too liberal!

A saner algorithm can be used by

```
{-# OPTIONS --without-K #-}
```

With or without K?

It turns out that there are models of dependent type theory in which UIP/K is not validated.

In fact, the default pattern matching algorithm of Agda is too liberal!

A saner algorithm can be used by

```
{-# OPTIONS --without-K #-}
```

The reason why it is not activated by default is that this makes proofs more complicated...

With or without K?

It turns out that there are models of dependent type theory in which UIP/K is not validated.

In fact, the default pattern matching algorithm of Agda is too liberal!

A saner algorithm can be used by

```
{-# OPTIONS --without-K #-}
```

The reason why it is not activated by default is that this makes proofs more complicated... but also more interesting!

With or without K?

It makes your life easier, but if you have too much of it you run into problems.

Types as spaces

In order to understand that, we should change our way of thinking:

0. in boolean logic: a type is either 0 or 1,

Types as spaces

In order to understand that, we should change our way of thinking:

0. in boolean logic: a type is either `0` or `1`,

1. in logic with UIP: a type is a set
(e.g. `Nat` can be seen as the set of natural numbers)

Types as spaces

In order to understand that, we should change our way of thinking:

0. in boolean logic: a type is either `0` or `1`,

1. in logic with UIP: a type is a set
(e.g. `Nat` can be seen as the set of natural numbers)

⋮

Types as spaces

In order to understand that, we should change our way of thinking:

0. in boolean logic: a type is either `0` or `1`,

1. in logic with UIP: a type is a set
(e.g. `Nat` can be seen as the set of natural numbers)

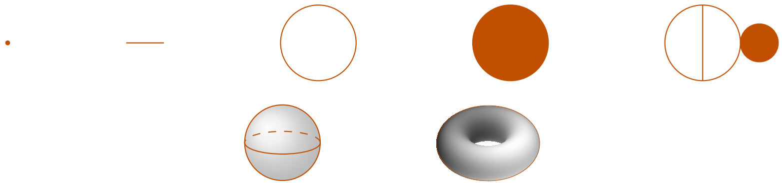
⋮

∞. in logic without UIP: a type is a **space**

Spaces

We will not precisely define what a space is, but you can think of it as

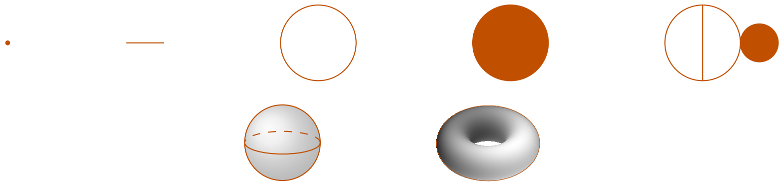
- a topological space, or
- something built up by gluing polyhedra (in arbitrary dimensions)



Spaces

We will not precisely define what a space is, but you can think of it as

- a topological space, or
- something built up by gluing polyhedra (in arbitrary dimensions)



considered up to "deformation".

Paths

We write I for the segment

$$I = [0, 1]$$

with the euclidean topology.

Paths

We write I for the segment

$$I = [0, 1]$$

with the euclidean topology.

A **path** from x to y in A is a continuous function

$$f : I \to A$$

such that $f(0) = x$ and $f(1) = y$.

Paths

We write I for the segment

$$I = [0, 1]$$

with the euclidean topology.

A **path** from x to y in A is a continuous function

$$f : I \to A$$

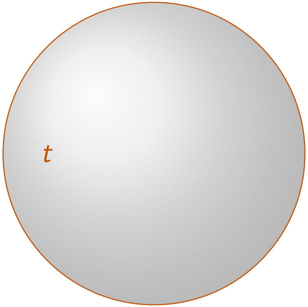
such that $f(0) = x$ and $f(1) = y$.

In particular, given a point $x \in A$, there is always the **constant path** from x to x, defined by $f(i) = x$ for $i \in I$.

Identity types in spaces

The idea is that an

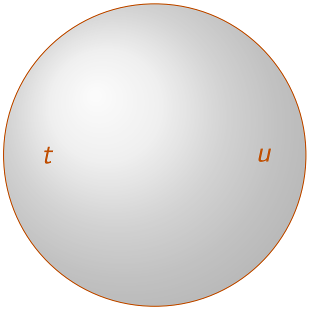
- a term $t : A$ corresponds to a point in the space



Identity types in spaces

The idea is that an

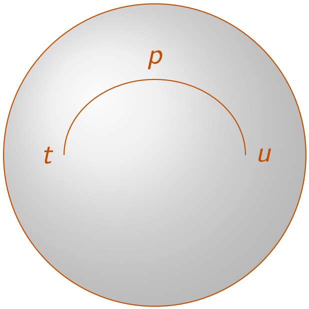
- a term $t : A$ corresponds to a point in the space



Identity types in spaces

The idea is that an

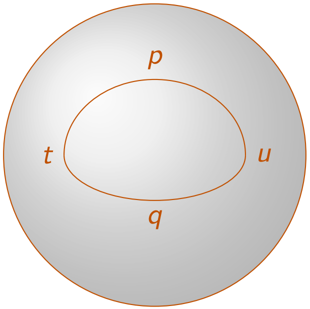
- a term $t : A$ corresponds to a point in the space
- an equality $p : \mathrm{Id}_A(t, u)$ is path between t and u



Identity types in spaces

The idea is that an

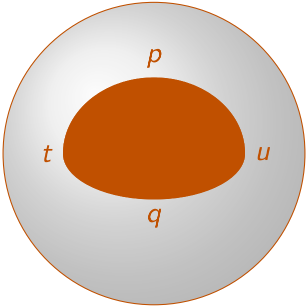
- a term $t : A$ corresponds to a point in the space
- an equality $p : \mathrm{Id}_A(t, u)$ is path between t and u



Identity types in spaces

The idea is that an

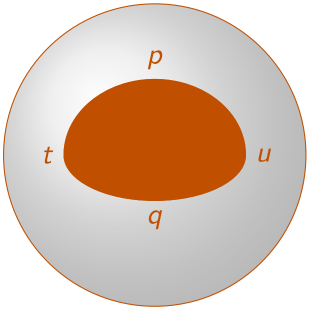
- a term $t : A$ corresponds to a point in the space
- an equality $p : \mathrm{Id}_A(t, u)$ is path between t and u
- an equality between equalities $\alpha : \mathrm{Id}_{\mathrm{Id}_A(t,u)}(p, q)$ is homotopy between p and q



Identity types in spaces

The idea is that an

- a term $t : A$ corresponds to a point in the space
- an equality $p : \mathsf{Id}_A(t, u)$ is path between t and u
- an equality between equalities $\alpha : \mathsf{Id}_{\mathsf{Id}_A(t,u)}(p, q)$ is homotopy between p and q
- and so on.



Homotopy equivalence

Two functions $f : A \to B$ and $g : B \to A$ between spaces are **homotopic** when

- $f(x)$ can be deformed into $g(x)$, i.e. there is a path from $f(x)$ to $g(x)$,
- in a way which is continuous in x.

We write this $f \sim g$.

Homotopy equivalence

Two functions $f : A \to B$ and $g : B \to A$ between spaces are **homotopic** when

- $f(x)$ can be deformed into $g(x)$, i.e. there is a path from $f(x)$ to $g(x)$,
- in a way which is continuous in x.

We write this $f \sim g$.

Two spaces A and B are **homotopy equivalent** when there is

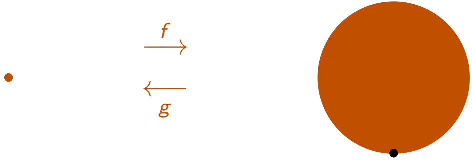
$$f : A \to B \qquad \text{and} \qquad g : B \to A$$

such that

$$g \circ f \sim \mathrm{id}_A \qquad \text{and} \qquad f \circ g \sim \mathrm{id}_B$$

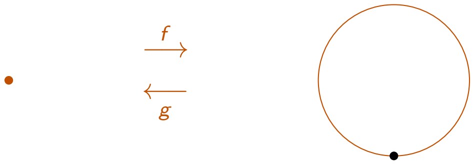
Homotopy equivalence

For instance, the following spaces are homotopy equivalent:



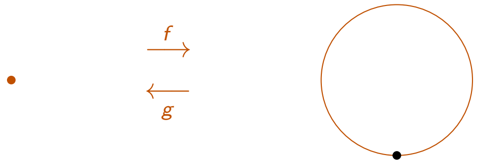
Homotopy equivalence

For instance, the following spaces are not homotopy equivalent:



Homotopy equivalence

For instance, the following spaces are not homotopy equivalent:



It can be shown that equivalent spaces always have the same number of "holes" in every dimension (and this can even be taken as a definition).

Homotopy type theory

Because it considers spaces up to homotopy equivalence, the resulting theory is called **homotopy type theory**.

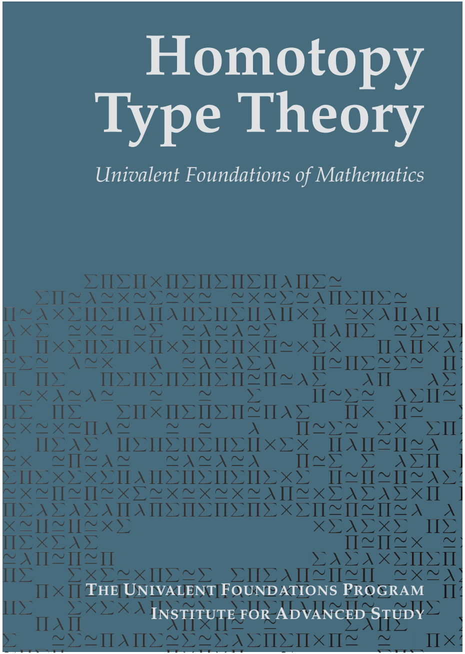
This point of view was introduced by Voevodsky (and other people) around 2006.

Homotopy type theory

Because it considers spaces up to homotopy equivalence, the resulting theory is called **homotopy type theory**.

This point of view was introduced by Voevodsky (and other people) around 2006.

In order to make this clear, we write `Type` instead of `Set` in the following.


Homotopy
Type Theory
Univalent Foundations of Mathematics
The Univalent Foundations Program
Institute for Advanced Study

Elimination on identity types

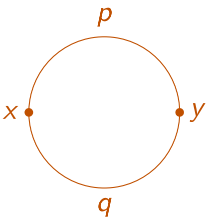
The elimination principle says that in order to show property depending on a path p, it is enough to show it for the constant path `refl`.

If we consider spaces up to homotopy we should be careful!

Elimination on identity types

The elimination principle says that in order to show property depending on a path p, it is enough to show it for the constant path `refl`.

If we consider spaces up to homotopy we should be careful!

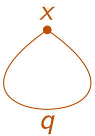



```
UIP-proof : {A : Type} (x y : A) (p q : x ≡ y) → p ≡ q
UIP-proof x y p q = ?
```

Elimination on identity types

The elimination principle says that in order to show property depending on a path p, it is enough to show it for the constant path `refl`.

If we consider spaces up to homotopy we should be careful!




```
UIP-proof : {A : Type} (x y : A) (p q : x ≡ y) → p ≡ q
UIP-proof x .x refl q = ?
```

Elimination on identity types

The elimination principle says that in order to show property depending on a path p, it is enough to show it for the constant path `refl`.

If we consider spaces up to homotopy we should be careful!


```
UIP-proof : {A : Type} (x y : A) (p q : x ≡ y) → p ≡ q
UIP-proof x .x refl refl = ?
```

Elimination on identity types

The elimination principle says that in order to show property depending on a path p, it is enough to show it for the constant path `refl`.

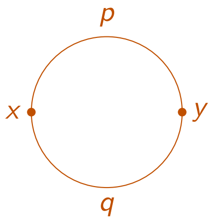
If we consider spaces up to homotopy we should be careful!


```
UIP-proof : {A : Type} (x y : A) (p q : x ≡ y) → p ≡ q
UIP-proof x .x refl refl = refl
```

Elimination on identity types

The elimination principle says that in order to show property depending on a path p, it is enough to show it for the constant path `refl`.

If we consider spaces up to homotopy we should be careful!



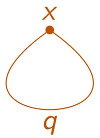

```
{-# OPTIONS --without-K #-}

UIP-proof : {A : Type} (x y : A) (p q : x ≡ y) → p ≡ q
UIP-proof x y p q = ?
```

Elimination on identity types

The elimination principle says that in order to show property depending on a path p, it is enough to show it for the constant path `refl`.

If we consider spaces up to homotopy we should be careful!




```
{-# OPTIONS --without-K #-}

UIP-proof : {A : Type} (x y : A) (p q : x ≡ y) → p ≡ q
UIP-proof x .x refl q = ?
```

Elimination on identity types

The elimination principle says that in order to show property depending on a path *p*, it is enough to show it for the constant path `refl`.

If we consider spaces up to homotopy we should be careful!

```
I'm not sure if there should be a case for the constructor refl,
because I get stuck when trying to solve the following unification
problems (inferred index =? expected index):
  x₁ =? x₁
Possible reason why unification failed:
  Cannot eliminate reflexive equation x₁ = x₁ of type A₁ because K
  has been disabled.
when checking that the expression ? has type refl ≡ q
```

The structure of paths

Three important constructions on paths:

- we can build the constant path on a point (`refl`),

The structure of paths

Three important constructions on paths:

- we can build the constant path on a point (`refl`),
- we can concatenate paths:

The structure of paths

Three important constructions on paths:

- we can build the constant path on a point (`refl`),
- we can concatenate paths:

```
_•_ : {x y z : A} → (p : x ≡ y) → (q : y ≡ z) → x ≡ z
refl • q = q
```

The structure of paths

Three important constructions on paths:

- we can build the constant path on a point (`refl`),
- we can concatenate paths:

```
_•_ : {x y z : A} → (p : x ≡ y) → (q : y ≡ z) → x ≡ z
refl • q = q
```

- we can compute the inverse of a path:

The structure of paths

Three important constructions on paths:

- we can build the constant path on a point (`refl`),
- we can concatenate paths:

  ```
  _•_ : {x y z : A} → (p : x ≡ y) → (q : y ≡ z) → x ≡ z
  refl • q = q
  ```

- we can compute the inverse of a path:

  ```
  ! : {x y : A} → x ≡ y → y ≡ x
  ! refl = refl
  ```

The structure of paths

Moreover,

- concatenation is associative:

The structure of paths

Moreover,

- concatenation is associative:

```
•-assoc : {x y z w : A} →
          (p : x ≡ y) → (q : y ≡ z) → (r : z ≡ w) →
          (p • q) • r ≡ p • (q • r)
•-assoc refl refl refl = refl
```

The structure of paths

Moreover,

- concatenation is associative:

```
•-assoc : {x y z w : A} →
          (p : x ≡ y) → (q : y ≡ z) → (r : z ≡ w) →
          (p • q) • r ≡ p • (q • r)
•-assoc refl refl refl = refl
```

- admits constant paths as neutral elements,

The structure of paths

Moreover,

- concatenation is associative:

```
•-assoc : {x y z w : A} →
          (p : x ≡ y) → (q : y ≡ z) → (r : z ≡ w) →
          (p • q) • r ≡ p • (q • r)
•-assoc refl refl refl = refl
```

- admits constant paths as neutral elements,
- and inverses act as such:

The structure of paths

Moreover,

- concatenation is associative:

```
•-assoc : {x y z w : A} →
          (p : x ≡ y) → (q : y ≡ z) → (r : z ≡ w) →
          (p • q) • r ≡ p • (q • r)
•-assoc refl refl refl = refl
```

- admits constant paths as neutral elements,
- and inverses act as such:

```
•-inv-l : {x y : A} → (p : x ≡ y) → ! p • p ≡ refl
•-inv-l refl = refl
•-inv-r : {x y : A} → (p : x ≡ y) → p • ! p ≡ refl
•-inv-r refl = refl
```

The structure of paths

This structure is like a group, excepting that we can only compose paths when their target and source endpoints match: this is called a **groupoid**.

The structure of paths

This structure is like a group, excepting that we can only compose paths when their target and source endpoints match: this is called a **groupoid**.

Moreover, note that the laws are not exactly satisfied: they are only so up to higher paths...

Part IV

n-types

Classifying types

Now that we have this idea that

$$\text{TYPE} = \text{SPACE}$$

we can begin to think of a classification of types.

Classifying types

Now that we have this idea that

$$\text{TYPE} = \text{SPACE}$$

and

$$\text{equality proof} = \text{path}$$

we can begin to think of a classification of types.

Propositions

The most simple kind of types are **propositions** which we can think of as being either

- true = a point (or at least equivalent to a point), or
- false = empty.

Propositions

The most simple kind of types are **propositions** which we can think of as being either

- true = a point (or at least equivalent to a point), or
- false = empty.

We can define the type of propositions as

Propositions

The most simple kind of types are **propositions** which we can think of as being either

- true = a point (or at least equivalent to a point), or
- false = empty.

We can define the type of propositions as

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

Propositions

For instance, the empty type is a proposition:

Propositions

For instance, the empty type is a proposition:

```
⊥-isProp : isProp ⊥
⊥-isProp ()
```

Propositions

For instance, the empty type is a proposition:

```
⊥-isProp : isProp ⊥
⊥-isProp ()
```

The unit type is also a proposition:

Propositions

For instance, the empty type is a proposition:

```
⊥-isProp : isProp ⊥
⊥-isProp ()
```

The unit type is also a proposition:

```
⊤-isProp : isProp ⊤
⊤-isProp tt tt = refl
```

Propositions

For instance, the empty type is a proposition:

```
⊥-isProp : isProp ⊥
⊥-isProp ()
```

The unit type is also a proposition:

```
⊤-isProp : isProp ⊤
⊤-isProp tt tt = refl
```

But the booleans are not a proposition:

Propositions

For instance, the empty type is a proposition:

```
⊥-isProp : isProp ⊥
⊥-isProp ()
```

The unit type is also a proposition:

```
⊤-isProp : isProp ⊤
⊤-isProp tt tt = refl
```

But the booleans are not a proposition:

```
Bool-isnt-prop : ¬ (isProp Bool)
Bool-isnt-prop P with P true false
Bool-isnt-prop P | ()
```

Propositions

We can even define the type of all propositions as

Propositions

We can even define the type of all propositions as

```
PROP : Type
PROP = Σ Type isProp
```

Propositions

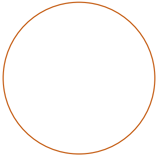
We can even define the type of all propositions as

```
PROP : Type
PROP = Σ Type isProp
```

(we are really ignoring universes here)

Propositions

A first, it might seem that the circle

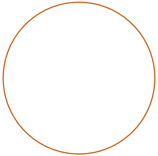


is a proposition

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

Propositions

A first, it might seem that the circle



is a proposition

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

but it is not so, because all the functions we can write are continuous!

Propositions

Propositions act very much like sets with 0 or 1 elements.

Propositions

Propositions act very much like sets with 0 or 1 elements.

(up to some approximation because they are not classical!)

Propositions

Propositions act very much like sets with 0 or 1 elements.
(up to some approximation because they are not classical!)

For instance, the product (= conjunction) of two such sets is also such:

	0	1
0	0	0
1	0	1

Propositions

Propositions act very much like sets with 0 or 1 elements.
(up to some approximation because they are not classical!)

For instance, the product (= conjunction) of two such sets is also such:

	0	1
0	0	0
1	0	1

Similarly, for any set A, the set $A \to 0$ of functions (= implications) contains either 1 or 0 elements (depending on whether A is empty or not). We thus expect $\neg A$ to be a proposition for any A.

Propositions

We can show that the conjunction of two propositions is a proposition:

Propositions

We can show that the conjunction of two propositions is a proposition:

```
isProp-∧ : {A B : Type} → isProp A → isProp B → isProp (A × B)
isProp-∧ PA PB (a , b) (a' , b') with PA a a' , PB b b'
isProp-∧ PA PB (a , b) (.a , .b) | refl , refl = refl
```

Propositions

We can show that the conjunction of two propositions is a proposition:

```
isProp-∧ : {A B : Type} → isProp A → isProp B → isProp (A × B)
isProp-∧ PA PB (a , b) (a' , b') with PA a a' , PB b b'
isProp-∧ PA PB (a , b) (.a , .b) | refl , refl = refl
```

However, we cannot prove that $\neg A$ is a proposition

Propositions

We can show that the conjunction of two propositions is a proposition:

```
isProp-∧ : {A B : Type} → isProp A → isProp B → isProp (A × B)
isProp-∧ PA PB (a , b) (a' , b') with PA a a' , PB b b'
isProp-∧ PA PB (a , b) (.a , .b) | refl , refl = refl
```

However, we cannot prove that $\neg A$ is a proposition

```
isProp-¬ : {A : Type} → isProp (¬ A)
isProp-¬ ¬x ¬y = ?
```

Propositions

We can show that the conjunction of two propositions is a proposition:

```
isProp-∧ : {A B : Type} → isProp A → isProp B → isProp (A × B)
isProp-∧ PA PB (a , b) (a' , b') with PA a a' , PB b b'
isProp-∧ PA PB (a , b) (.a , .b) | refl , refl = refl
```

However, we cannot prove that $\neg A$ is a proposition

```
isProp-¬ : {A : Type} → isProp (¬ A)
isProp-¬ ¬x ¬y = ?
```

because we do not have any useful tool to show the equality of functions.

Function extensionality

In fact, we need **function extensionality**:

```
postulate funext : {A : Type} {B : A → Type} → {f g : (x : A) → B x} →
                   ((x : A) → f x ≡ g x) → f ≡ g
```

We already mentioned that this axiom was not reasonable, because we want to capture intensional properties of functions.

Function extensionality

In fact, we need **function extensionality**:

```
postulate funext : {A : Type} {B : A → Type} → {f g : (x : A) → B x} →
                   ((x : A) → f x ≡ g x) → f ≡ g
```

We already mentioned that this axiom was not reasonable, because we want to capture intensional properties of functions.

However, in a homotopic setting, this does not say that `f` is the same as `g`, only that one can be deformed to the other.

Function extensionality

In fact, we need **function extensionality**:

```
postulate funext : {A : Type} {B : A → Type} → {f g : (x : A) → B x} →
                   ((x : A) → f x ≡ g x) → f ≡ g
```

We already mentioned that this axiom was not reasonable, because we want to capture intensional properties of functions.

However, in a homotopic setting, this does not say that `f` is the same as `g`, only that one can be deformed to the other.

Moreover, we will see that it actually follows from the main (only?) axiom of homotopy type theory: *univalence*.

Propositions

We can now show that $\neg A$ is a proposition:

Propositions

We can now show that $\neg A$ is a proposition:

```
isProp-¬ : {A : Type} → isProp (¬ A)
isProp-¬ ¬x ¬y = funext (λ x → ⊥-elim (¬x x))
```

Propositions

The fact of being a proposition is itself a proposition:

```
isProp-isProp : {A : Type} → isProp (isProp A)
```

(set later on for the proof).

Propositions

Note : we started Curry-Howard as

$$\text{propositions} = \text{types}$$

but what we really have is

$$\text{propositions} \subseteq \text{types}$$

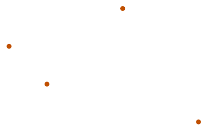
Sets

The next thing we can define are **sets**.

Sets

The next thing we can define are **sets**.

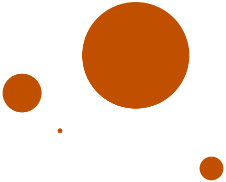
We should follow the idea that a set consists of points:



Sets

The next thing we can define are **sets**.

We should follow the idea that a set consists of points up to homotopy:

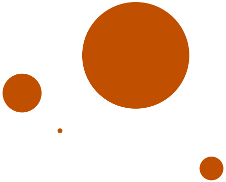


(typically the circle is not a set).

Sets

The next thing we can define are **sets**.

We should follow the idea that a set consists of points up to homotopy:



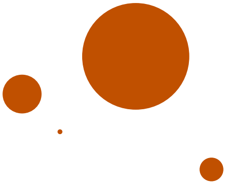
(typically the circle is not a set). We therefore define

```
isSet : Type → Type
isSet A = (x y : A) → (p q : x ≡ y) → p ≡ q
```

Sets

The next thing we can define are **sets**.

We should follow the idea that a set consists of points up to homotopy:



(typically the circle is not a set). We therefore define

```
isSet : Type → Type
isSet A = (x y : A) → isProp (x ≡ y)
```

Sets

For instance, booleans form a set:

Sets

For instance, booleans form a set:

```
Bool-isSet : isSet Bool
Bool-isSet false .false refl refl = refl
Bool-isSet true .true refl refl = refl
```

Sets

For instance, booleans form a set:

```
Bool-isSet : isSet Bool
Bool-isSet false .false refl refl = refl
Bool-isSet true .true refl refl = refl
```

as well as natural numbers:

Sets

For instance, booleans form a set:

```
Bool-isSet : isSet Bool
Bool-isSet false .false refl refl = refl
Bool-isSet true .true refl refl = refl
```

as well as natural numbers:

```
suc-≡ : {m n : ℕ} → (p : suc m ≡ suc n) →
        Σ (m ≡ n) (λ q → cong suc q ≡ p)
suc-≡ refl = refl , refl

ℕ-isSet : isSet ℕ
ℕ-isSet zero .zero refl refl = refl
ℕ-isSet (suc x) .(suc x) refl p with (suc-≡ p)
... | q , e = trans (cong (cong suc) (ℕ-isSet x x refl q)) e
```

More generally,

Theorem (Hedberg)
Any type with a decidable equality is a set.

Sets

Also, propositions are sets:

```
aProp-isSet : {A : Type} → isProp A → isSet A
```

We can notice a "pattern" (of length one...): a set is a type in which there is a at most one equality (up to homotopy) between two elements:

```
isSet : Type → Type
isSet A = (x y : A) → isProp (x ≡ y)
```

We can notice a "pattern" (of length one...): a set is a type in which there is a at most one equality (up to homotopy) between two elements:

```
isSet : Type → Type
isSet A = (x y : A) → isProp (x ≡ y)
```

We therefore define a 1-type as

1-types

We can notice a "pattern" (of length one...): a set is a type in which there is a at most one equality (up to homotopy) between two elements:

```
isSet : Type → Type
isSet A = (x y : A) → isProp (x ≡ y)
```

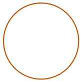
We therefore define a 1-type as

```
is1Type : Type → Type
is1Type A = (x y : A) → isSet (x ≡ y)
```

1-types

A 1-type is type in which there is at most one equality between two equalities:

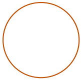
- the circle is a 1-type:



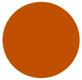
1-types

A 1-type is type in which there is at most one equality between two equalities:

- the circle is a 1-type:



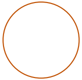
- the disk is a 1-type:



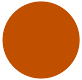
1-types

A 1-type is type in which there is at most one equality between two equalities:

- the circle is a 1-type:



- the disk is a 1-type:



- the sphere is not a 1-type:

n-types

From now on, the definition of n-types is clear:

- a 0-type is a set (by convention),
- an $(n+1)$-type is a type in which $\mathtt{x} \equiv \mathtt{y}$ is an n-type for every elements $\mathtt{x}$ and $\mathtt{y}$.

n-types

From now on, the definition of n-types is clear:

- a 0-type is a set (by convention),
- an $(n+1)$-type is a type in which `x ≡ y` is an n-type for every elements `x` and `y`.

We have seen that a (-1)-type is a proposition:

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

n-types

From now on, the definition of n-types is clear:

- a 0-type is a set (by convention),
- an $(n+1)$-type is a type in which `x ≡ y` is an n-type for every elements `x` and `y`.

We have seen that a (-1)-type is a proposition:

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

An (-2)-type is

n-types

From now on, the definition of n-types is clear:

- a 0-type is a set (by convention),
- an $(n+1)$-type is a type in which `x ≡ y` is an n-type for every elements `x` and `y`.

We have seen that a (-1)-type is a proposition:

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

An (-2)-type is a contractible type:

```
isContr : Type → Type
isContr A = Σ A (λ x → (y : A) → x ≡ y)
```

n-types

From now on, the definition of n-types is clear:

- a 0-type is a set (by convention),
- an $(n+1)$-type is a type in which `x ≡ y` is an n-type for every elements `x` and `y`.

We have seen that a (-1)-type is a proposition:

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

An (-2)-type is a contractible type:

```
isContr : Type → Type
isContr A = Σ A (λ x → (y : A) → x ≡ y)
```

A (-3)-type is

n-types

From now on, the definition of n-types is clear:

- a 0-type is a set (by convention),
- an $(n+1)$-type is a type in which `x ≡ y` is an n-type for every elements `x` and `y`.

We have seen that a (-1)-type is a proposition:

```
isProp : Type → Type
isProp A = (x y : A) → x ≡ y
```

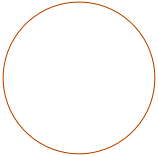
An (-2)-type is a contractible type:

```
isContr : Type → Type
isContr A = Σ A (λ x → (y : A) → x ≡ y)
```

A (-3)-type is a contractible type.

(-2)-types

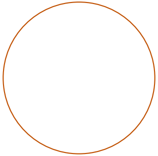
Again, it might seem that the circle



is a (-2)-type

```
isContr : Type → Type
isContr A = Σ A (λ x → (y : A) → x ≡ y)
```

Again, it might seem that the circle



is a (-2)-type

```
isContr : Type → Type
isContr A = Σ A (λ x → (y : A) → x ≡ y)
```

but it is not so because functions have to be continuous.

n-types

In Agda, we can thus define (starting at 0 instead of -2):

In Agda, we can thus define (starting at 0 instead of -2):

```
hasLevel : ℕ → Type → Type
hasLevel  zero    A = isContr A
hasLevel (suc n) A = (x y : A) → hasLevel n (x ≡ y)
```

n-types

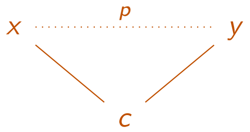
We can show interesting properties such as:

Theorem
Any n-type is an $(n+1)$-type.

We can show interesting properties such as:

Theorem
Any n-type is an $(n+1)$-type.

```
cumulative : (n : ℕ) {A : Type} → hasLevel n A → hasLevel (suc n) A
cumulative zero L x y =
  (! (snd L x) • snd L y) , λ { refl → •-inv-l (snd L x) }
cumulative (suc n) L x y = cumulative n (L x y)
```



n-types

Or that

Theorem
Being an n-type is a proposition.

Or that

Theorem
Being an n-type is a proposition.

For instance,

```
isProp-isProp : {A : Type} → isProp (isProp A)
isProp-isProp {A = A} f g =
  funext2 {f = f} {g = g} (λ x y → aProp-isSet g x y (f x y) (g x y))
```

where `funext2` is function extensionality for functions with two arguments.

Part V

Univalence

Let's see some other operations available with paths.

Application

Functions respect identities (intuitively, because they are continuous):

Application

Functions respect identities (intuitively, because they are continuous):

```
ap : {A B : Type} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
ap f refl = refl
```

In other words, we can **apply** a function to a path.

Application

Functions respect identities (intuitively, because they are continuous):

```
ap : {A B : Type} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
ap f refl = refl
```

In other words, we can **apply** a function to a path.

This is what we called `cong` before.

Lemma
Application is compatible with concatenation.

Application

Lemma
Application is compatible with concatenation.

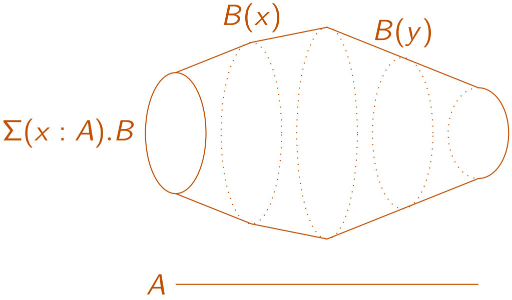
Proof.

```
•-ap : {A B : Type} {x y z : A} → (f : A → B) →
       (p : x ≡ y) → (q : y ≡ z) → ap f (p • q) ≡ ap f p • ap f q
•-ap f refl q = refl
```

□

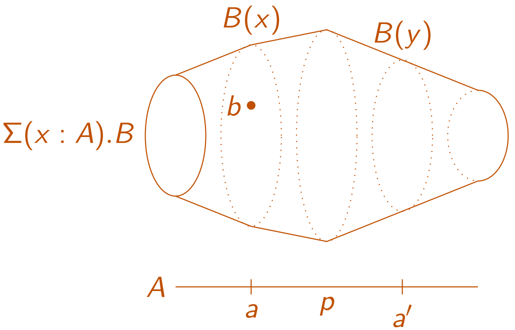
Type families

A type family `P : A → Type` should be thought of as a collection of spaces `P a` for each `a : A` which varies *continuously* in `a`:



Type families

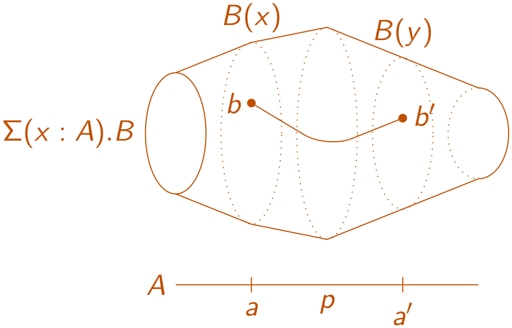
A type family `P : A → Type` should be thought of as a collection of spaces `P a` for each `a : A` which varies *continuously* in `a`:



Given a path p in A from a to a' and a point b in $P\ a$

Type families

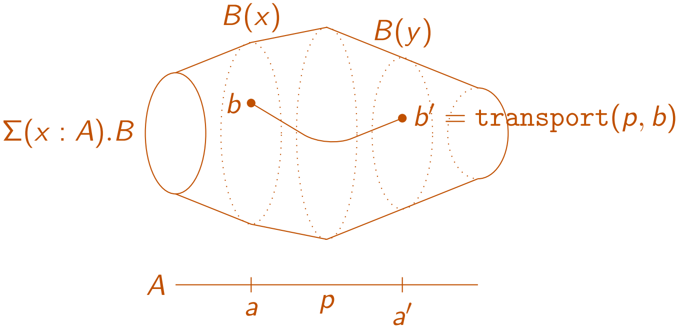
A type family `P : A → Type` should be thought of as a collection of spaces `P a` for each `a : A` which varies *continuously* in `a`:



Given a path p in A from a to a' and a point b in $P\,a$ we expect that there is a unique path in P whose "projection" on A is p.

Type families

A type family `P : A → Type` should be thought of as a collection of spaces `P a` for each `a : A` which varies *continuously* in `a`:



Given a path p in A from a to a' and a point b in $P\ a$ we expect that there is a unique path in P whose "projection" on A is p.

We call its other end in $P\ b$, the **transport** of b along p.

Transport

Formally, the **transport** operation is defined as

Transport

Formally, the **transport** operation is defined as

```
transport : {A : Type} {x y : A} (P : A → Type) → x ≡ y → P x → P y
transport P refl x = x
```

Transport

Formally, the **transport** operation is defined as

```
transport : {A : Type} {x y : A} (P : A → Type) → x ≡ y → P x → P y
transport P refl x = x
```

This is what we called `subst` before.

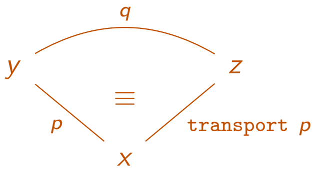
Transport

We can show that transporting a path along one of its end amounts to composing it with the path:

Transport

We can show that transporting a path along one of its end amounts to composing it with the path:

```
transport-≡-r : {A : Type} {x y z : A} → (p : x ≡ y) (q : y ≡ z) →
                transport (λ y → x ≡ y) q p ≡ (p • q)
transport-≡-r p refl = sym (•-unit-r p)
```



Transport

We can show that transporting a path along one of its end amounts to composing it with the path:

```
transport-≡-r : {A : Type} {x y z : A} → (p : x ≡ y) (q : y ≡ z) →
                transport (λ y → x ≡ y) q p ≡ (p • q)
transport-≡-r p refl = sym (•-unit-r p)
```

and similarly on the other side:

```
transport-≡-l : {A : Type} {x y z : A} → (p : x ≡ y) (q : y ≡ z) →
                transport (λ y → y ≡ z) (! p) q ≡ (p • q)
transport-≡-l refl q = refl
```

Dependent application

We have defined application of a function $f : A \to B$ to a path $p : x \equiv y$ in A:

```
ap : {A B : Type} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
```

Dependent application

We have defined application of a function $f : A \to B$ to a path $p : x \equiv y$ in A:

```
ap : {A B : Type} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
```

We would like to generalize this operation to a dependent function $f : (a : A) \to B(a)$.

Dependent application

We have defined application of a function $f : A \to B$ to a path $p : x \equiv y$ in A:

```
ap : {A B : Type} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
```

We would like to generalize this operation to a dependent function $f : (a : A) \to B(a)$.

We are thus tempted to prove

```
apd : {A : Type} {B : A → Type} {x y : A} →
      (f : (a : A) → B a) → x ≡ y → f x ≡ f y
```

Dependent application

We have defined application of a function $f : A \to B$ to a path $p : x \equiv y$ in A:

```
ap : {A B : Type} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
```

We would like to generalize this operation to a dependent function $f : (a : A) \to B(a)$.

We are thus tempted to prove

```
apd : {A : Type} {B : A → Type} {x y : A} →
      (f : (a : A) → B a) → x ≡ y → f x ≡ f y
```

What is the problem?

Dependent application

We have defined application of a function $f : A \to B$ to a path $p : x \equiv y$ in A:

```
ap : {A B : Type} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
```

We would like to generalize this operation to a dependent function $f : (a : A) \to B(a)$.

We are thus tempted to prove

```
apd : {A : Type} {B : A → Type} {x y : A} →
      (f : (a : A) → B a) → x ≡ y → f x ≡ f y
```

What is the problem?

The correct definition of **dependent application** is

```
apd : {A : Type} {B : A → Type} {x y : A} → (f : (x : A) → B x) →
      (p : x ≡ y) → transport B p (f x) ≡ f y
apd f refl = refl
```

Dependent application

Lemma
Every proposition A *is a set:*

Dependent application

Lemma
Every proposition A is a set:

```
aProp-isSet : {A : Type} → isProp A → isSet A
aProp-isSet {A} P x y p q = ?
```

Dependent application

Lemma
Every proposition A is a set:

```
aProp-isSet : {A : Type} → isProp A → isSet A
aProp-isSet {A} P x .x refl q = ?
```

Dependent application

Lemma
Every proposition A is a set:

```
aProp-isSet : {A : Type} → isProp A → isSet A
aProp-isSet {A} P x .x refl refl = ?
```

Dependent application

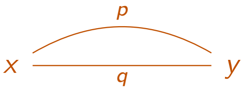
Lemma
Every proposition A is a set:

```
aProp-isSet : {A : Type} → isProp A → isSet A
aProp-isSet {A} P x .x refl refl = ?
```

```
I'm not sure if there should be a case for the constructor refl,
because I get stuck when trying to solve the following unification
problems (inferred index =? expected index):
  x₁ =? x₁
Possible reason why unification failed:
  Cannot eliminate reflexive equation x₁ = x₁ of type A₁ because K
  has been disabled.
when checking that the expression ? has type refl ≡ q
```

Dependent application

Lemma
Every proposition A is a set:

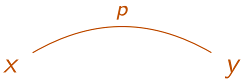


Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$.

Dependent application

Lemma
Every proposition A is a set:

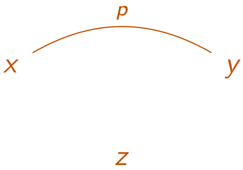


Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p

Dependent application

Lemma
Every proposition A is a set:

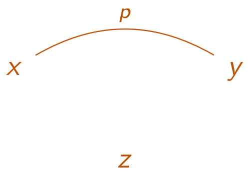


Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p and take a point z.

Dependent application

Lemma
Every proposition A is a set:

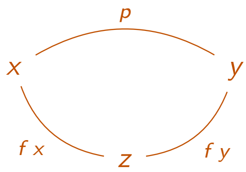


Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p and take a point z.
Since A is a proposition, we have a function $f : (x : A) \to z \equiv x$.

Dependent application

Lemma
Every proposition A is a set:

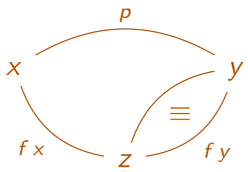


Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p and take a point z. Since A is a proposition, we have a function $f : (x : A) \to z \equiv x$. In particular, we can consider $f\ x$ and $f\ y$.

Dependent application

Lemma
Every proposition A is a set:

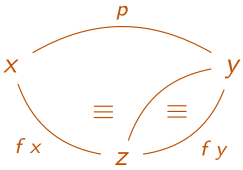


Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p and take a point z. Since A is a proposition, we have a function $f : (x : A) \to z \equiv x$. In particular, we can consider $f\ x$ and $f\ y$. Using `apd` of f to p we get a path from `transport` $(f\ x)\ p$ to $f\ y$

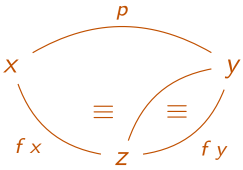
Dependent application

Lemma
Every proposition A is a set:



Proof.
Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p and take a point z. Since A is a proposition, we have a function $f : (x : A) \to z \equiv x$. In particular, we can consider $f\, x$ and $f\, y$. Using $\texttt{apd}$ of f to p we get a path from $\texttt{transport}\,(f\, x)\, p$ to $f\, y$ but the first is equal to $f\, x \cdot p$.

Lemma
Every proposition A is a set:

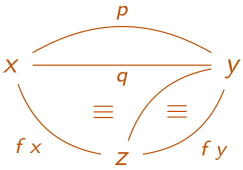


Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p and take a point z. Since A is a proposition, we have a function $f : (x : A) \to z \equiv x$. In particular, we can consider $f\ x$ and $f\ y$. Using $\texttt{apd}$ of f to p we get a path from $\texttt{transport}\,(f\ x)\,p$ to $f\ y$ but the first is equal to $f\ x \cdot p$. Therefore, $f\ x \cdot p \equiv f\ y$, i.e. $p \equiv (f\ x)^{-1} \cdot f\ y$.

Dependent application

Lemma
Every proposition A is a set:

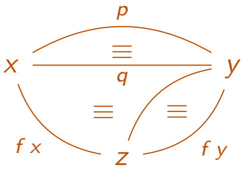


Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p and take a point z. Since A is a proposition, we have a function $f : (x : A) \to z \equiv x$. In particular, we can consider $f\ x$ and $f\ y$. Using $\texttt{apd}$ of f to p we get a path from $\texttt{transport}\,(f\ x)\,p$ to $f\ y$ but the first is equal to $f\ x \cdot p$. Therefore, $f\ x \cdot p \equiv f\ y$, i.e. $p \equiv (f\ x)^{-1} \cdot f\ y$. Similarly, $q \equiv (f\ x)^{-1} \cdot f\ y$,

Dependent application

Lemma
Every proposition A is a set:



Proof.

Given two paths $p, q : x \equiv y$, we have to show $p \equiv q$. Consider p and take a point z. Since A is a proposition, we have a function $f : (x : A) \to z \equiv x$. In particular, we can consider $f\ x$ and $f\ y$. Using $\texttt{apd}$ of f to p we get a path from $\texttt{transport}\ (f\ x)\ p$ to $f\ y$ but the first is equal to $f\ x \cdot p$. Therefore, $f\ x \cdot p \equiv f\ y$, i.e. $p \equiv (f\ x)^{-1} \cdot f\ y$. Similarly, $q \equiv (f\ x)^{-1} \cdot f\ y$, and finally $p \equiv q$. □

Dependent application

Note that in order to show that $p \equiv (f\,x)^{-1} \cdot f\,y$, we could also have done an induction on p and shown the result in the case where p is `refl`, i.e.

$$\texttt{refl} \equiv (f\,x)^{-1} \cdot f\,x$$

which we have already shown.

Dependent application

Note that in order to show that $p \equiv (f\,x)^{-1} \cdot f\,y$, we could also have done an induction on p and shown the result in the case where p is `refl`, i.e.

$$\texttt{refl} \equiv (f\,x)^{-1} \cdot f\,x$$

which we have already shown.

Formally,

```
aProp-isSet : {A : Type} → isProp A → isSet A
aProp-isSet {A} P x y p q = trans (lem x p) (sym (lem x q))
  where
  lem : (z : A) (p : x ≡ y) → p ≡ ! (P z x) • (P z y)
  lem z refl = sym (•-inv-l (P z x))
```

Homotopy

Two functions are **homotopic** when they are extensionally equal:

```
_∼_ : {A : Type} {B : A → Type} (f g : (x : A) → B x) → Type
_∼_ {A} f g = (x : A) → f x ≡ g x
```

This relation is different from equality between functions
(if we do not assume function extensionality or some other axiom).

Equivalences

We can define the identity function on a type A by

Equivalences

We can define the identity function on a type A by

```
id : {A : Type} → A → A
id x = x
```

Equivalences

We can define the identity function on a type A by

```
id : {A : Type} → A → A
id x = x
```

We can define the composition of functions by

Equivalences

We can define the identity function on a type A by

```
id : {A : Type} → A → A
id x = x
```

We can define the composition of functions by

```
_∘_ : {A : Type} {B : Type} {C : Type} → (B → C) → (A → B) → (A → C)
(g ∘ f) x = g (f x)
```

Equivalences

A function is an **equivalence** when

Equivalences

A function is an **equivalence** when

```
isEquiv : {A : Type} {B : Type} → (A → B) → Type
isEquiv {A} {B} f =
  Σ (B → A) (λ g → (f ∘ g) ∼ id) × Σ (B → A) (λ g → (g ∘ f) ∼ id)
```

Equivalences

A function is an **equivalence** when

```
isEquiv : {A : Type} {B : Type} → (A → B) → Type
isEquiv {A} {B} f =
  Σ (B → A) (λ g → (f ∘ g) ∼ id) × Σ (B → A) (λ g → (g ∘ f) ∼ id)
```

The type of equivalences between two types is

Equivalences

A function is an **equivalence** when

```
isEquiv : {A : Type} {B : Type} → (A → B) → Type
isEquiv {A} {B} f =
  Σ (B → A) (λ g → (f ∘ g) ∼ id) × Σ (B → A) (λ g → (g ∘ f) ∼ id)
```

The type of equivalences between two types is

```
_≃_ : (A B : Type) → Type
A ≃ B = Σ (A → B) isEquiv
```

Equivalences

It seems that we could have defined equivalences more simply as

```
isEquiv' : {A : Type} {B : Type} → (A → B) → Type
isEquiv' {A} {B} f = Σ (B → A) (λ g → (f ∘ g) ∼ id × (g ∘ f) ∼ id)
```

but this is not equivalent to the previous definition.

Equivalences

It seems that we could have defined equivalences more simply as

```
isEquiv' : {A : Type} {B : Type} → (A → B) → Type
isEquiv' {A} {B} f = Σ (B → A) (λ g → (f ∘ g) ∼ id × (g ∘ f) ∼ id)
```

but this is not equivalent to the previous definition.

In fact this not the right definition, one way to see this is that we have

```
isEquiv-isProp : {A : Type} {B : Type} (f : A → B) → isProp (isEquiv f)
```

but there exists a function `f : A → B` such that this does not hold for `isEquiv’`.

Univalence

It is easy to show that any two equal types are equivalent:

Univalence

It is easy to show that any two equal types are equivalent:

```
id-to-equiv : {A B : Type} → (A ≡ B) → (A ≃ B)
id-to-equiv refl = id , ((id , (λ _ → refl)) , id , (λ _ → refl))
```

Univalence

It is easy to show that any two equal types are equivalent:

```
id-to-equiv : {A B : Type} → (A ≡ B) → (A ≃ B)
id-to-equiv refl = id , ((id , (λ _ → refl)) , id , (λ _ → refl))
```

The **univalence axiom** says that this map is itself an equivalence:

```
postulate univalence : (A B : Type) → isEquiv (id-to-equiv {A} {B})
```

Univalence

It is easy to show that any two equal types are equivalent:

```
id-to-equiv : {A B : Type} → (A ≡ B) → (A ≃ B)
id-to-equiv refl = id , ((id , (λ _ → refl)) , id , (λ _ → refl))
```

The **univalence axiom** says that this map is itself an equivalence:

```
postulate univalence : (A B : Type) → isEquiv (id-to-equiv {A} {B})
```

Note that we need to be serious about (cumulative) universes here since we have an equivalence between a small and a big type.

Univalence

The most useful consequence of this is that we have a map

```
ua : {A B : Type} → (A ≃ B) → (A ≡ B)
ua {A} {B} f with univalence A B
ua {A} {B} f | (g , _) , _ = g f
```

which allows us to make an equality from an equivalence, for which we have the usual tools such as transport.

Unary and binary natural numbers

For instance, we can define binary natural numbers as

Unary and binary natural numbers

For instance, we can define binary natural numbers as

```
data Bin : Set where
  b0 : Bin
  b1 : List Bool → Bin
```

(the second being `1` followed by a reversed list of bits).

Unary and binary natural numbers

For instance, we can define binary natural numbers as

```
data Bin : Set where
  b0 : Bin
  b1 : List Bool → Bin
```

(the second being `1` followed by a reversed list of bits).

We can convert binary numbers into unary ones:

Unary and binary natural numbers

For instance, we can define binary natural numbers as

```
data Bin : Set where
  b0 : Bin
  b1 : List Bool → Bin
```

(the second being 1 followed by a reversed list of bits).

We can convert binary numbers into unary ones:

```
Bin-to-Nat : Bin → ℕ
Bin-to-Nat b0 = 0
Bin-to-Nat (b1 []) = 1
Bin-to-Nat (b1 (x ∷ l)) = (if x then 1 else 0) + 2 * Bin-to-Nat (b1 l)
```

Unary and binary natural numbers

This function can be shown to induce an equivalence between the two representations:

```
Bin-to-Nat-isEquiv : isEquiv (Bin-to-Nat)
```

Unary and binary natural numbers

This function can be shown to induce an equivalence between the two representations:

```
Bin-to-Nat-isEquiv : isEquiv (Bin-to-Nat)
```

We can therefore define addition on binary numbers from the one on unary numbers:

This function can be shown to induce an equivalence between the two representations:

```
Bin-to-Nat-isEquiv : isEquiv (Bin-to-Nat)
```

We can therefore define addition on binary numbers from the one on unary numbers:

```
add : Bin → Bin → Bin
add = transport
        (λ A → (A → A → A))
        (sym (ua (Bin-to-Nat , Bin-to-Nat-isEquiv)))
        _+_
```

Univalence

In order to understand better univalence, it is simpler to take a variant of

```
id-to-equiv : {A B : Type} → (A ≡ B) → (A ≃ B)
id-to-equiv refl = id , ((id , (λ _ → refl)) , id , (λ _ → refl))
```

when defining

```
postulate univalence : (A B : Type) → isEquiv (id-to-equiv {A} {B})
```

Univalence

Any path $p : A \equiv B$ induces a **coercion** function

Univalence

Any path $p : A \equiv B$ induces a **coercion** function

```
coe : {A B : Type} → (A ≡ B) → A → B
coe p x = transport (λ A → A) p x
```

Univalence

Any path $p : A \equiv B$ induces a **coercion** function

```
coe : {A B : Type} → (A ≡ B) → A → B
coe p x = transport (λ A → A) p x
```

which is easily seen to be an equivalence

Univalence

Any path $p : A \equiv B$ induces a **coercion** function

```
coe : {A B : Type} → (A ≡ B) → A → B
coe p x = transport (λ A → A) p x
```

which is easily seen to be an equivalence

```
coe-isEquiv : {A B : Type} (p : A ≡ B) → isEquiv (coe p)
coe-isEquiv refl = (id , (λ x → refl)) , (id , λ x → refl)
```

Univalence

Any path $p : A \equiv B$ induces a **coercion** function

```
coe : {A B : Type} → (A ≡ B) → A → B
coe p x = transport (λ A → A) p x
```

which is easily seen to be an equivalence

```
coe-isEquiv : {A B : Type} (p : A ≡ B) → isEquiv (coe p)
coe-isEquiv refl = (id , (λ x → refl)) , (id , λ x → refl)
```

from which we define

```
id-to-equiv : {A B : Type} → (A ≡ B) → (A ≃ B)
id-to-equiv p = coe p , coe-isEquiv p
```

Univalence

Any path $p : A \equiv B$ induces a **coercion** function

```
coe : {A B : Type} → (A ≡ B) → A → B
coe p x = transport (λ A → A) p x
```

which is easily seen to be an equivalence

```
coe-isEquiv : {A B : Type} (p : A ≡ B) → isEquiv (coe p)
coe-isEquiv refl = (id , (λ x → refl)) , (id , λ x → refl)
```

from which we define

```
id-to-equiv : {A B : Type} → (A ≡ B) → (A ≃ B)
id-to-equiv p = coe p , coe-isEquiv p
```

and

```
postulate univalence : (A B : Type) → isEquiv (id-to-equiv {A} {B})
```

Univalence

The univalence axiom thus says that the function (**elimination** rule)

$$\mathsf{coe} : (A \equiv B) \to (A \simeq B)$$

Univalence

The univalence axiom thus says that the function (**elimination** rule)

$$\texttt{coe} : (A \equiv B) \to (A \simeq B)$$

admits an “inverse” (**introduction** rule)

$$\texttt{ua} : (A \simeq B) \to (A \equiv B)$$

i.e.

Univalence

The univalence axiom thus says that the function (**elimination** rule)

$$\mathsf{coe} : (A \equiv B) \to (A \simeq B)$$

admits an "inverse" (**introduction** rule)

$$\mathtt{ua} : (A \simeq B) \to (A \equiv B)$$

i.e.

- **computation** rule: for every equivalence $f : A \to B$ and $x : A$

$$\mathsf{coe}\,(\mathtt{ua}\,f)\,x \equiv f\,x$$

Univalence

The univalence axiom thus says that the function (**elimination** rule)

$$\mathtt{coe} : (A \equiv B) \to (A \simeq B)$$

admits an "inverse" (**introduction** rule)

$$\mathtt{ua} : (A \simeq B) \to (A \equiv B)$$

i.e.

- **computation** rule: for every equivalence $f : A \to B$ and $x : A$

$$\mathtt{coe}\,(\mathtt{ua}\,f)\,x \equiv f\ x$$

- **uniqueness** rule: for every $p : A \equiv B$,

$$\mathtt{ua}\,(\mathtt{coe}\,p) \equiv p$$

Univalence and classical logic

The univalence axiom is incompatible with classical logic: if we suppose that

$$\neg\neg A \to A$$

holds for every type A then we can prove $\bot$.

Univalence and classical logic

The univalence axiom is incompatible with classical logic: if we suppose that

$$\neg\neg A \to A$$

holds for every type A then we can prove $\bot$.

The general idea of the proof is as follows:

Univalence and classical logic

The univalence axiom is incompatible with classical logic: if we suppose that

$$\neg\neg A \to A$$

holds for every type A then we can prove $\bot$.

The general idea of the proof is as follows:

- we have an equivalence $f : \mathsf{Bool} \simeq \mathsf{Bool}$ exchanging `true` and `false`,

Univalence and classical logic

The univalence axiom is incompatible with classical logic: if we suppose that

$$\neg\neg A \to A$$

holds for every type A then we can prove $\bot$.

The general idea of the proof is as follows:

- we have an equivalence $f : \mathsf{Bool} \simeq \mathsf{Bool}$ exchanging $\texttt{true}$ and $\texttt{false}$,
- by univalence, it induces a non-trivial path $p : \mathsf{Bool} \equiv \mathsf{Bool}$,

Univalence and classical logic

The univalence axiom is incompatible with classical logic: if we suppose that

$$\neg\neg A \to A$$

holds for every type A then we can prove $\bot$.

The general idea of the proof is as follows:

- we have an equivalence $f : \mathsf{Bool} \simeq \mathsf{Bool}$ exchanging `true` and `false`,
- by univalence, it induces a non-trivial path $p : \mathsf{Bool} \equiv \mathsf{Bool}$,
- the map $\neg\neg A \to A$ amounts to choosing and element a of A,

Univalence and classical logic

The univalence axiom is incompatible with classical logic: if we suppose that

$$\neg\neg A \to A$$

holds for every type A then we can prove $\bot$.

The general idea of the proof is as follows:

- we have an equivalence $f : \mathsf{Bool} \simeq \mathsf{Bool}$ exchanging $\mathtt{true}$ and $\mathtt{false}$,
- by univalence, it induces a non-trivial path $p : \mathsf{Bool} \equiv \mathsf{Bool}$,
- the map $\neg\neg A \to A$ amounts to choosing and element a of A,
- by transport and happly we can show that we should have $a \cong \mathtt{not}\ a$,

Univalence and classical logic

The univalence axiom is incompatible with classical logic: if we suppose that

$$\neg\neg A \to A$$

holds for every type A then we can prove $\bot$.

The general idea of the proof is as follows:

- we have an equivalence $f : \mathrm{Bool} \simeq \mathrm{Bool}$ exchanging $\mathtt{true}$ and $\mathtt{false}$,
- by univalence, it induces a non-trivial path $p : \mathrm{Bool} \equiv \mathrm{Bool}$,
- the map $\neg\neg A \to A$ amounts to choosing and element a of A,
- by transport and happly we can show that we should have $a \cong \mathtt{not}\, a$,
- we therefore have $\mathtt{true} \equiv \mathtt{false}$ from which we can deduce $\bot$.

Univalence and classical logic

However, there is no contradiction in supposing

$$\neg\neg A \to A$$

for every proposition A.

Higher inductive types

For now, we don't have many non-trivial 2-types at our disposal (excepting SET).

Namely, all the types we constructed up to now are sets
(natural numbers, lists over sets, etc.).

For instance, there is no easy way to construct something which looks like a circle.

Higher inductive types

For now, we don't have many non-trivial 2-types at our disposal (excepting `SET`).

Namely, all the types we constructed up to now are sets
(natural numbers, lists over sets, etc.).

For instance, there is no easy way to construct something which looks like a circle.

In order to do so, we need a generalization of inductive types: **higher inductive types**.

Higher inductive types

In an inductive type, we specify constructors which add elements to the type.

Higher inductive types

In an inductive type, we specify constructors which add elements to the type.

In a higher inductive type, we can also add identities between the elements of the type.

Higher inductive types

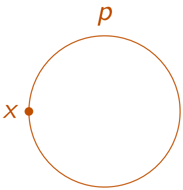
In an inductive type, we specify constructors which add elements to the type.

In a higher inductive type, we can also add identities between the elements of the type.

Those are not completely well understood (and implemented) as of now.

Higher inductive types

For instance, we can define the circle

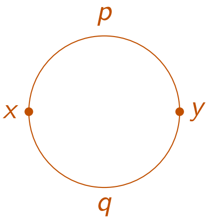


as

```
data Circle : Type where
  x : Circle
  p : x ≡ x
```

Higher inductive types

For instance, we can define the circle



as

```
data Circle : Type where
  x : Circle
  y : Circle
  p : x ≡ y
  q : x ≡ y
```

Higher inductive types

Recall that for booleans

```
data Bool : Type where
  true  : Bool
  false : Bool
```

the recursion principle is that given

- a type A,
- an element $t : A$
- an element $u : A$

there exists a unique function

$$f : \mathrm{Bool} \to A$$

such that $f\,\texttt{true} = t$ and $f\,\texttt{false} = u$.

Higher inductive types

Recall that for booleans

```
data Bool : Type where
  true  : Bool
  false : Bool
```

the induction principle is that given

- a predicate $P : \text{Bool} \to \text{Type}$,
- an element $t : P\,\texttt{true}$
- an element $u : P\,\texttt{false}$

there exists a unique function

$$f : (x : \text{Bool}) \to P\,x$$

such that $f\,\texttt{true} = t$ and $f\,\texttt{false} = u$.

Higher inductive types

Recall that for booleans

```
data Bool : Type where
  true  : Bool
  false : Bool
```

the induction principle is that given

- a predicate $P : \mathsf{Bool} \to \mathsf{Type}$,
- an element $t : P\,\texttt{true}$
- an element $u : P\,\texttt{false}$

there exists a unique function

$$f : (x : \mathsf{Bool}) \to P\,x$$

such that $f\,\texttt{true} = t$ and $f\,\texttt{false} = u$.
Exercise: define `not`.

Higher inductive types

If we consider the type

```
data Circle : Type where
  base : Circle
  loop : base ≡ base
```

the corresponding induction principle is that given

- a predicate $P : \mathtt{Circle} \to \mathsf{Type}$,
- an element $b : P\,\mathtt{base}$,
- a path $l : P\,\mathtt{base} \equiv P\,\mathtt{base}$

there exists a (unique up to homotopy) function

$$f : (x : \mathtt{Circle}) \to P\,x$$

such that $f\,\mathtt{base} = b$ and $f\,\mathtt{loop} = l$.

Suspension

The **suspension** of a type *A* is

```
data Susp (A : Type) : Type where
  N : Susp A
  S : Susp B
  p : (x : A) → N ≡ S
```

Suspension

The **suspension** of a type A is

```
data Susp (A : Type) : Type where
  N : Susp A
  S : Susp B
  p : (x : A) → N ≡ S
```

Starting with the empty type $\bot$, its suspension is $\mathbb{S}^0$

Suspension

The **suspension** of a type A is

```
data Susp (A : Type) : Type where
  N : Susp A
  S : Susp B
  p : (x : A) → N ≡ S
```

Starting with the empty type $\bot$, its suspension is $\mathbb{S}^0$

Suspension

The **suspension** of a type A is

```
data Susp (A : Type) : Type where
  N : Susp A
  S : Susp B
  p : (x : A) → N ≡ S
```

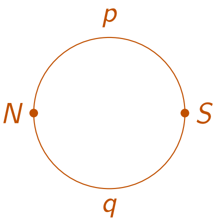
Starting with $\mathbb{S}^0$, its suspension is the circle

Suspension

The **suspension** of a type A is

```
data Susp (A : Type) : Type where
  N : Susp A
  S : Susp B
  p : (x : A) → N ≡ S
```

Starting with $\mathbb{S}^0$, its suspension is the circle

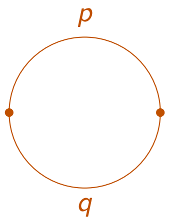


Suspension

The **suspension** of a type *A* is

```
data Susp (A : Type) : Type where
  N : Susp A
  S : Susp B
  p : (x : A) → N ≡ S
```

Starting with the circle,

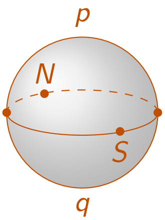


Suspension

The **suspension** of a type *A* is

```
data Susp (A : Type) : Type where
  N : Susp A
  S : Susp B
  p : (x : A) → N ≡ S
```

Starting with the circle, its suspension is the sphere



Suspension

The **suspension** of a type *A* is

```
data Susp (A : Type) : Type where
  N : Susp A
  S : Susp B
  p : (x : A) → N ≡ S
```

In this way, we can construct the *n*-sphere for any dimension *n*...

Propositional truncation

The **propositional truncation** of a type is

```
data Trunc (A : Type) : Type where
  carrier : A → Trunc A
  trivial : (x y : Trunc A) → x ≡ y
```

Propositional truncation

The **propositional truncation** of a type is

```
data Trunc (A : Type) : Type where
  carrier : A → Trunc A
  trivial : (x y : Trunc A) → x ≡ y
```

Lemma
For every type A, `Trunc` A *is a proposition.*

Propositional truncation

The **propositional truncation** of a type is

```
data Trunc (A : Type) : Type where
  carrier : A → Trunc A
  trivial : (x y : Trunc A) → x ≡ y
```

Lemma
For every type A, `Trunc` A *is a proposition.*

It can be thought of as turning a type A into a proposition,
i.e. it is (equivalent to) a point when A is non-empty and a empty when A is empty.

Propositional truncation

The **propositional truncation** of a type is

```
data Trunc (A : Type) : Type where
  carrier : A → Trunc A
  trivial : (x y : Trunc A) → x ≡ y
```

Lemma
For every type A, `Trunc` *A is a proposition.*

It can be thought of as turning a type A into a proposition,
i.e. it is (equivalent to) a point when A is non-empty and a empty when A is empty.

However, this is done in an intuitionistic way (the above would rather be $\neg\neg A$).

Propositional truncation

The recursion principle says that given

- a type B,
- a function $g : A \to B$,
- a path $x \equiv y$ for every $x, y : B$,

there exists a unique function

$$f : \mathtt{Trunc}\, A \to B$$

such that $f\,x = g\,x$ for $x : A$ and given $x, y : A$, $\mathrm{ap}\, f$ sends the specified path from x to y in A to the one between $f\,x$ and $f\,y$ in B.

Propositional truncation

For instance, there is a canonical map from $\texttt{Trunc}\, A$ to $\neg\neg A$ induced by

- the map $A \to \neg\neg A$ sending x to $\lambda f. f\, x$,
- the fact that $\neg\neg A$ is a proposition.

Propositional truncation

For instance, there is a canonical map from $\texttt{Trunc}\,A$ to $\neg\neg A$ induced by

- the map $A \to \neg\neg A$ sending x to $\lambda f . f\,x$,
- the fact that $\neg\neg A$ is a proposition.

It is an equivalence if and only if the logic is classical.