

# Higher Inductive Types

*Hype for Types Guest Lecture*

*December 1, 2024*

**Runming Li**

# Inductive types are great

```
data ℕ : Type where  
  zero : ℕ  
  suc  : ℕ → ℕ
```

```
data List (A : Type) : Type where  
  [] : List A  
  _::_ : A → List A → List A
```

# Program with inductive types

```
data  $\mathbb{Z}$  : Type where  
  Pos :  $\mathbb{N} \rightarrow \mathbb{Z}$   
  Neg :  $\mathbb{N} \rightarrow \mathbb{Z}$ 
```

# Program with inductive types

```
data  $\mathbb{Z}$  : Type where
```

```
  Pos :  $\mathbb{N} \rightarrow \mathbb{Z}$ 
```

```
  Neg :  $\mathbb{N} \rightarrow \mathbb{Z}$ 
```

Invariant: Pos zero should always be equal to Neg zero.

# Program with invariants in mind is hard

$\text{pred} : \mathbb{Z} \rightarrow \mathbb{Z}$

$\text{pred} (\text{Pos } \text{zero}) = \text{Neg } (\text{suc } \text{zero})$

$\text{pred} (\text{Pos } (\text{suc } x)) = \text{Pos } x$

$\text{pred} (\text{Neg } x) = \text{Neg } (\text{suc } x)$

# Program with invariants in mind is hard

$\text{pred} : \mathbb{Z} \rightarrow \mathbb{Z}$

$\text{pred} (\text{Pos } \text{zero}) = \text{Neg } (\text{suc } \text{zero})$

$\text{pred} (\text{Pos } (\text{suc } x)) = \text{Pos } x$

$\text{pred} (\text{Neg } x) = \text{Neg } (\text{suc } x)$

Convince yourself that this function respects the invariant:

$\text{pred} (\text{Pos } \text{zero})$  should be equal to  $\text{pred} (\text{Neg } \text{zero})$ .

# What if I made a mistake?

pred\_bad :  $\mathbb{Z} \rightarrow \mathbb{Z}$

pred\_bad (Pos zero) = Neg zero -- bug here!

pred\_bad (Pos (suc x)) = Pos x

pred\_bad (Neg x) = Neg (suc x)

# What if I made a mistake?

`pred_bad :  $\mathbb{Z} \rightarrow \mathbb{Z}$`

`pred_bad (Pos zero) = Neg zero -- bug here!`

`pred_bad (Pos (suc x)) = Pos x`

`pred_bad (Neg x) = Neg (suc x)`

The invariant is broken:

`pred_bad (Pos zero) = Neg zero` is not equal to  
`pred_bad (Neg zero) = Neg (suc zero)`.

Nevertheless this program still typechecks.



“If it typechecks, it is correct” is a lie.

# Let's make the invariant part of the type

```
data  $\mathbb{Z}'$  : Type where  
  Pos :  $\mathbb{N} \rightarrow \mathbb{Z}'$   
  Neg :  $\mathbb{N} \rightarrow \mathbb{Z}'$   
  Inv : Pos zero  $\equiv$  Neg zero
```

# Let's make the invariant part of the type

```
data  $\mathbb{Z}'$  : Type where  
  Pos :  $\mathbb{N} \rightarrow \mathbb{Z}'$   
  Neg :  $\mathbb{N} \rightarrow \mathbb{Z}'$   
  Inv : Pos zero  $\equiv$  Neg zero
```

Now when we program with  $\mathbb{Z}'$ , we need to consider three cases: Pos, Neg, and Inv.

# Typechecker checks the invariant for us

$\text{pred\_bad}' : \mathbb{Z}' \rightarrow \mathbb{Z}'$

$\text{pred\_bad}' (\text{Pos zero}) = \text{Neg zero}$

$\text{pred\_bad}' (\text{Pos (suc } x)) = \text{Pos } x$

$\text{pred\_bad}' (\text{Neg } x) = \text{Neg (suc } x)$

$\text{pred\_bad}' (\text{Inv } i) = \{\! \! \! \}$

# Typechecker checks the invariant for us

$\text{pred\_bad}' : \mathbb{Z}' \rightarrow \mathbb{Z}'$

$\text{pred\_bad}' (\text{Pos zero}) = \text{Neg zero}$

$\text{pred\_bad}' (\text{Pos (suc } x)) = \text{Pos } x$

$\text{pred\_bad}' (\text{Neg } x) = \text{Neg (suc } x)$

$\text{pred\_bad}' (\text{Inv } i) = \{! \ !\}$

What should we fill in the hole?

# Typechecker checks the invariant for us

```
pred_bad' : ℤ' → ℤ'  
pred_bad' (Pos zero) = Neg zero  
pred_bad' (Pos (suc x)) = Pos x  
pred_bad' (Neg x) = Neg (suc x)  
pred_bad' (Inv i) = {! !}
```

What should we fill in the hole?

Let's see what does the typechecker want:

```
---- Boundary (wanted) -----  
i = i0 ⊢ Neg zero  
i = i1 ⊢ Neg (suc zero)
```

# Fill in the hole

$\text{pred}' : \mathbb{Z}' \rightarrow \mathbb{Z}'$

$\text{pred}' (\text{Pos zero}) = \text{Neg (suc zero)}$

$\text{pred}' (\text{Pos (suc } x)) = \text{Pos } x$

$\text{pred}' (\text{Neg } x) = \text{Neg (suc } x)$

$\text{pred}' (\text{Inv } i) = \text{refl } \{x = \text{Neg (suc zero)}\} i$

$\text{refl}$  means “reflexivity” of equality, which is a proof that  $x$  is equal to  $x$  for any  $x$ .

# Fill in the hole

$\text{pred}' : \mathbb{Z}' \rightarrow \mathbb{Z}'$

$\text{pred}' (\text{Pos zero}) = \text{Neg (suc zero)}$

$\text{pred}' (\text{Pos (suc } x)) = \text{Pos } x$

$\text{pred}' (\text{Neg } x) = \text{Neg (suc } x)$

$\text{pred}' (\text{Inv } i) = \text{refl } \{x = \text{Neg (suc zero)}\} i$

`refl` means “reflexivity” of equality, which is a proof that  $x$  is equal to  $x$  for any  $x$ .

Now the program typechecks, because we give a **proof** that `pred'` respects the invariant.



If it typechecks, it is correct.

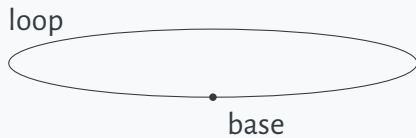
# Higher inductive types

Like inductive types, HIT has constructors such as **Pos** and **Neg**.

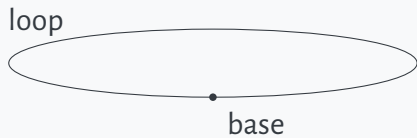
Unlike inductive types, HIT has extra constructors that introduce equalities, such as **Inv**.

# Circle

Classically, a circle is:  
 $\{(x, y) \mid x^2 + y^2 = r^2\}$ .



# Circle

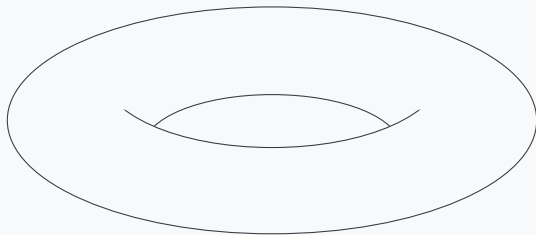


Classically, a circle is:  
 $\{(x, y) \mid x^2 + y^2 = r^2\}$ .

Using HIT, a circle is:

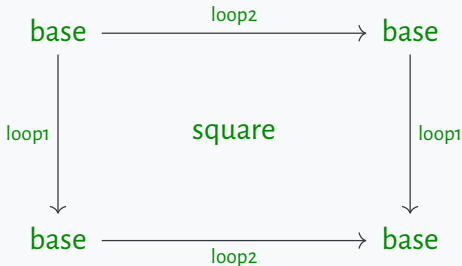
```
data Circle : Type where  
  base : Circle  
  loop : base ≡ base
```

How about a donut (torus)?



# Torus

```
data Torus : Type where  
  base : Torus  
  loop1 : base ≡ base  
  loop2 : base ≡ base  
  square : Square loop1 loop2 loop2 loop1
```



# A torus is two circles

*In topology, a ring torus is homeomorphic to the Cartesian product of two circles.*

*(Wikipedia)*

# A torus is two circles

*In topology, a ring torus is homeomorphic to the Cartesian product of two circles.*

*(Wikipedia)*

$$\text{Torus} \simeq \text{Circle} \times \text{Circle} : \text{Torus} \equiv (\text{Circle} \times \text{Circle})$$

Proof by **induction** on the torus and the circles.



# Conclusion

- HIT rises from Homotopy Type Theory/Univalent Foundations (HoTT/UF).
- HIT is great for programming with invariants.
- HIT is great for proving mathematical theorems, especially in homotopy theory.