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 ℤ : Type where
  Pos : ℕ → ℤ
  Neg : ℕ → ℤ
```

Program with inductive types

```
data ℤ : Type where
  Pos : ℕ → ℤ
  Neg : ℕ → ℤ
```

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

Program with invariants in mind is hard

```
pred : ℤ → ℤ
pred (Pos zero) = Neg (suc zero)
pred (Pos (suc x)) = Pos x
pred (Neg x) = Neg (suc x)
```

Program with invariants in mind is hard

$$\begin{aligned}
&\mathsf{pred} : \mathbb{Z} \to \mathbb{Z}\\
&\mathsf{pred}\ (\mathsf{Pos\ zero}) = \mathsf{Neg}\ (\mathsf{suc\ zero})\\
&\mathsf{pred}\ (\mathsf{Pos}\ (\mathsf{suc}\ x)) = \mathsf{Pos}\ x\\
&\mathsf{pred}\ (\mathsf{Neg}\ x) = \mathsf{Neg}\ (\mathsf{suc}\ x)
\end{aligned}$$

Convince yourself that this function respects the invariant:

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

What if I made a mistake?

```
pred_bad : ℤ → 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 : ℤ → 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 ℤ' : Type where
  Pos : ℕ → ℤ'
  Neg : ℕ → ℤ'
  Inv : Pos zero ≡ Neg zero
```

Let's make the invariant part of the type

```
data ℤ' : Type where
  Pos : ℕ → ℤ'
  Neg : ℕ → ℤ'
  Inv : Pos zero ≡ Neg zero
```

Now when we program with ℤ', we need to consider three cases: Pos, Neg, and Inv.

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) = {!  !}
```

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?

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

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

refl means "reflexivity" of equality, which is a proof that x is equal to x for any x.

Fill in the hole

```
pred' : ℤ' → ℤ'
pred' (Pos zero) = Neg (suc zero)
pred' (Pos (suc x)) = Pos x
pred' (Neg x) = Neg (suc x)
pred' (Inv i) = refl {x = 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.

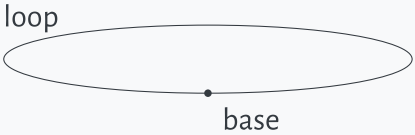
If it typechecks, it is correct.

Higher inductive types

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

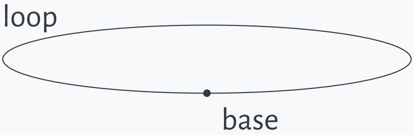
Unlike inductive types, HIT has extra constructors that introduce equalities, such as Inv.

Circle



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

Circle

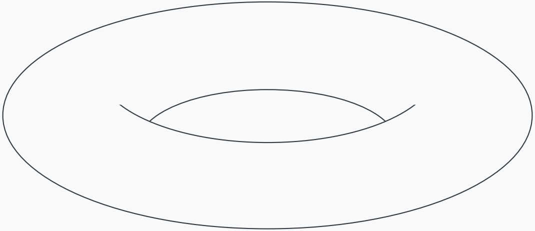


Classically, a circle is:
$\{(x, y) \mid x^2 + y^2 = r^2\}$.

Using HIT, a circle is:

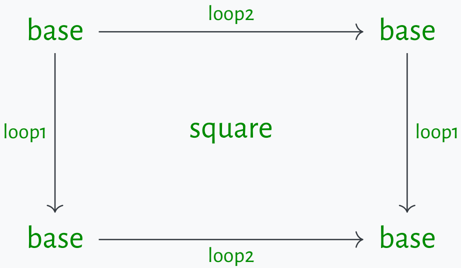
```
data Circle : Type where
  base : Circle
  loop : base ≡ base
```

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)*

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

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.