Postmodern Type Systems

Tesla Ice Zhang

About me

I am Tesla Ice Zhang. I work with programming languages.

Here's my profile: Tesla (Yinsen) Ice Zhang (psu.edu)

Dependent types

Let’s first dive into DT.

Popular type systems

- Assembly has no types.
- C, Java 4, C# 1, etc. have simple types.
- Java 5, C# 2, Kotlin, etc. have fancier types.
- C++ templates are even fancier.
- Swift, Haskell, etc. have some deductions.

We gradually improve the type system.

This allows us to type more values.

Functions

- The `printf` function. Can we check its arguments' types at compile time?

We first curry `printf : (string, any[]) -> ()`.

```
printf : string -> (any[] -> ())
```

That is to say,

```
printf("xyr") : any[] -> ()
printf("age %i") : any[] -> ()
printf("job %s") : any[] -> ()
printf("at (%f, %f)") : any[] -> ()
```

This is what we have:

```
printf("xyr") : any[] -> ()
printf("age %i") : any[] -> ()
printf("job %s") : any[] -> ()
printf("at (%f, %f)") : any[] -> ()
```

This is what we want:

```
printf("xyr") : () -> ()
printf("age %i") : (int) -> ()
printf("job %s") : (string) -> ()
printf("at (%f, %f)") : (float, float) -> ()
```

To do this, we need to change `printf`'s type into something else. What should we replace the `any[]` with?

```
printf : string -> (? -> ())
```

Observe: it depends on the first argument. So, let's invent this new syntax, which gives a name to the first argument, so we can talk about its value elsewhere in the type signature:

```
printf : (s : string) -> (? -> ())
```

Essentially, the ? should be a type calculated from s, so we replace it with a function. The

```
printf : (s : string) -> (? -> ())
```

Becomes:

```
printf : (s : string) -> (Fmt(s) -> ())
```

Observe `Fmt` – it should be a function, but what type does it have?

```
printf : (s : string) -> (Fmt(s) -> ())
```

It returns a type! What is the type of types? We don't know yet, but we can define it right now. Let's call it `Type`, so we have:

```
Fmt : string -> Type
```

Dependent types

- What we’ve just seen, is a dependent type system.
- It has functions returning types (in other words, type expressions with values inside), the type of types, etc.

Implementation

- A dependent type allows arbitrary mixture of types and values. So, it is natural to require the type checker to evaluate any code during type checking.
- We don't want the type checker to crash, so we require users to write code that never crashes (and never loops infinitely either).

Hard

Cubical type theory

—

A recent development of DT.

Equivalence relation

- We want to represent the equivalence relation as a type
- It should be something like

```
Eq : A -> A -> Type
```

- It needs to satisfy basic facts about equivalence, like substitution.
- How would it be implemented?

The 2-unit type

- We define the unit type (a type with only one canonical element) with two distinct constructors:

```
I : Type
left  : I
right : I
```

The 2-unit type

- Since it's the unit type, one is not allowed to distinguish between `left` and `right`.
- So, if we have `f : I -> A`, as it cannot tell which argument is actually passed to it, we can assume that `f(left)` should return the 'same' value as `f(right)`.

The relation

- Now, we define this function to construct an element of `Eq`. In other words, `Eq` is a wrapper of a function over `I`, and we can tell the value of `f(left)` and `f(right)` from the arguments of `Eq`.

```
path : (f : I -> A) -> Eq(f(left), f(right))
```

The relation

```
path : (f : I -> A) -> Eq(f(left), f(right))
```

- Let's use `path` to prove the reflexivity of `Eq`:

```
refl : (a : A) -> Eq(a, a)
refl(a) = path(i => a)
```

$$\forall a . a = a$$

Operations

```
path : (f : I -> A) -> Eq(f(left), f(right))
```

- We add an operation to elements of `Eq` that allows us to convert it back to a function using an (postfix) operator:

```
@ : Eq(a, b) -> (I -> A)
```

- We can also see it as an infix operator:

```
@ : Eq(a, b) -> I -> A
```

Operations

- Basic fact about `@`: if we have `f : Eq(a, b)`, then:
- `f @ left` will evaluate to `a`
- `f @ right` will evaluate to `b`

Function extensionality

- Let's use `path` to prove the extensionality of functions:

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(f, g)
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

Function extensionality

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(f, g)
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

Function extensionality

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(a => f(a), a => g(a))
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

Function extensionality

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(a => f(a), a => g(a))
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

Function extensionality

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(a => f(a), a => g(a))
funExt(f, g, p) = path(i => a => (p(a) @ i))

p(a)               : Eq(f(a), g(a))
```

Function extensionality

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(a => f(a), a => g(a))
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

`p(a)` : `Eq(f(a), g(a))`

`p(a) @ left` evaluates to `f(a)`

`p(a) @ right` evaluates to `g(a)`

Function extensionality

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(a => f(a), a => g(a))
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

```
p(a)               : Eq(f(a), g(a))
a => (p(a) @ left)
```
evaluates to `a => f(a)`

```
a => (p(a) @ right)
```
evaluates to `a => g(a)`

Function extensionality

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(a => f(a), a => g(a))
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

```
p(a)               : Eq(f(a), g(a))
```

`a => (p(a) @ left)` evaluates to `a => f(a)`

`a => (p(a) @ right)` evaluates to `a => g(a)`

Function extensionality

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(a => f(a), a => g(a))
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

`p(a)                : Eq(f(a), g(a))`

`a => (p(a) @ left)` evaluates to `a => f(a)`

`a => (p(a) @ right)` evaluates to `a => g(a)`

Function extensionality

`a => (p(a) @ left)` evaluates to `a => f(a)`
`a => (p(a) @ right)` evaluates to `a => g(a)`

Function extensionality

`a => (p(a) @ left)` evaluates to `a => f(a)`
`a => (p(a) @ right)` evaluates to `a => g(a)`

Observe `i => a => (p(a) @ i)`

Function extensionality

`a => (p(a) @ left)` evaluates to `a => f(a)`
`a => (p(a) @ right)` evaluates to `a => g(a)`

Observe `i => a => (p(a) @ i)`
This is a function that returns `a => f(a)` when applied `left`, and returns `a => g(a)` when applied `right`!

Function extensionality

Observe `i => a => (p(a) @ i)`
This is a function that returns `a => f(a)` when applied `left`,
and returns `a => g(a)` when applied `right`!

What would happen if we pass this function as an argument to:

```
path : (f : I -> A) -> Eq(f(left), f(right))
```

Function extensionality

Observe `i => a => (p(a) @ i)`
This is a function that returns `a => f(a)` when applied `left`, and returns `a => g(a)` when applied `right`!

```
path : (f : I -> A) -> Eq(f(left), f(right))
```

Function extensionality

Observe `i => a => (p(a) @ i)`
This is a function that returns `a => f(a)` when applied `left`,
and returns `a => g(a)` when applied `right`!

```
path : (f : I -> A) -> Eq(f(left), f(right))
```

Function extensionality

Observe `i => a => (p(a) @ i)`
This is a function that returns `a => f(a)` when applied `left`,
and returns `a => g(a)` when applied `right`!

```
path(i => a => (p(a) @ i))
  : Eq(a => f(a), a => g(a))
```

Function extensionality

- This is what we get eventually:

```
funExt : (f g : A -> B)
       -> ((a : A) -> Eq(f(a), g(a)))
       -> Eq(f, g)
funExt(f, g, p) = path(i => a => (p(a) @ i))
```

$$\forall f, g.(\forall a. f(a) = g(a)) \implies f = g$$

Simple exercise

- You can verify your understanding of `path` and `@` by implementing the following function:

```
pmap : (f : A -> B) -> Eq(a, b)
      -> Eq(f(a), f(b))
pmap(f, p) = ?
```

$$\forall f . a = b \implies f(a) = f(b)$$

Path as a type

- The definition of `Eq` (as a wrapper of `I -> A`) here is inspired from the notion of 'path' from topology.

> In mathematics, a **path** in a topological space X is a continuous function f from the unit interval $I = [0,1]$ to X
>
> $f: I \to X.$

Path as a type

- We call this type the `Path` type.
- `Path` in topology satisfies reflexivity, transitivity, symmetry, and substitution. It also enables various extensionality proofs, such as eta rules for coinductive types.
- We can prove some basic facts about topology in type systems with the `Path` type and *higher inductive types*.

Another operation

- The only one (primitive) operation we can do about `I`:

```
coe : (A : I -> Type) -> A(left)
    -> (i : I) -> A(i)
```

Generalized transport

```
coe : (A : I -> Type) -> A(left)
    -> (i : I) -> A(i)
```

- We can prove the substitution principle of `Eq` with `coe`.

What did we do?

—

What did people prove using type theories with the path type?

The theorem: $\pi_1(\mathbb{S}^1) \cong \mathbb{Z}$

We can prove a very basic fact, that the fundamental group of circle is isomorphic to the integer additive group, using **a type system**!

```
ΩS¹IsoInt : Iso ΩS¹ Int
Iso.fun ΩS¹IsoInt      = winding
Iso.inv ΩS¹IsoInt      = intLoop
Iso.rightInv ΩS¹IsoInt = windingIntLoop
Iso.leftInv ΩS¹IsoInt  = decodeEncode base
```

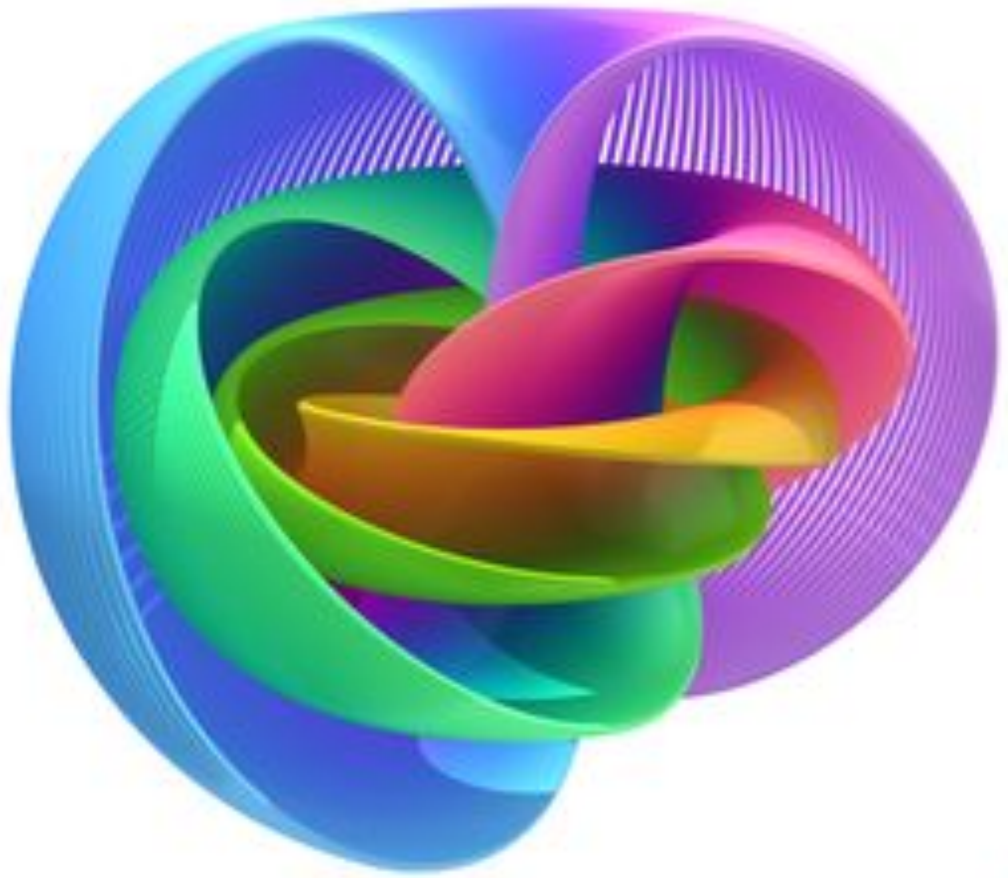
Klein-Bottles

Klein bottles are just torus with the surface twisted:

```
data Torus : Type where
  point : Torus
  line1 : point ≡ point
  line2 : point ≡ point
  square : PathP (λ i → line1 i ≡ line1 i) line2 line2

data KleinBottle : Type where
  point : KleinBottle
  line1 : point ≡ point
  line2 : point ≡ point
  square : PathP (λ i → line1 (~ i) ≡ line1 i) line2 line2
```

Hopf fibrations



Hopf fibrations, sphere as base space:

```
rotIsEquiv : (a : S¹) → isEquiv (a ·_)

HopfS² : S² → Type₀
HopfS² base = S¹
HopfS² (surf i j) = Glue S¹ (λ { (i = i0) → _ , idEquiv S¹
                                ; (i = i1) → _ , idEquiv S¹
                                ; (j = i0) → _ , idEquiv S¹
                                ; (j = i1) → _ , _ , rotIsEquiv (loop i) } )
```

Implementations

- The observational type theory by Conor McBride has a cubical version, called XTT.
- JetBrains created Arend, a programming language based on homotopy type theory.
- Agda supports cubical type theory as an extension.
- There is a book for mathematicians to study (homotopy) type theory: the HoTT book.

Formalizations

- The Grothendieck group has been formalized using cubical type theory.
- The 4-th homotopy group of 3-spheres has been formalized in Agda and cubical type theory.
- The Blakers-Massey theorem has been formalized in Arend (by JetBrains).
- The quotient set type and the finite multiset type can been defined as a higher inductive type.

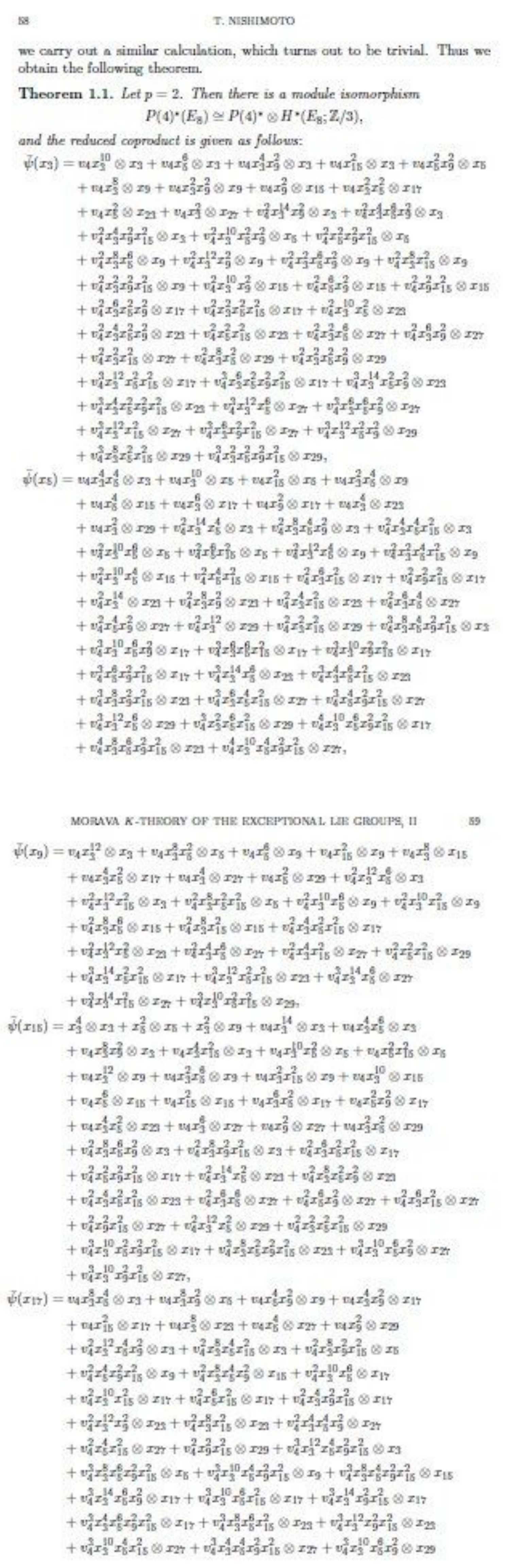
Why types?

So – what’s the point of all of these?

Why are we encoding things into types?

—

- Types and values can be checked by a computer (quickly), but a proof has to be checked by a mathematician (maybe takes a week, maybe with fee).
- We trust computers better than human on inspecting details.


58 T. NISHIMOTO

we carry out a similar calculation, which turns out to be trivial. Thus we obtain the following theorem.

Theorem 1.1. *Let $p = 2$. Then there is a module isomorphism*

$$P(4)^*(E_8) \cong P(4)^* \otimes H^*(E_8; \mathbb{Z}/3),$$

and the reduced coproduct is given as follows:

MORAVA K-THEORY OF THE EXCEPTIONAL LIE GROUPS, II 59

Why are we encoding things into types?

—

- How do we teach students ‘theorem proving’? What’s a valid proof and what’s not?
- The theory of types answers this *very clearly*: a proof is an instance of the type corresponds to the theorem.

Why are we encoding things into types?

—

- How do we study a mathematical concept? By asking the person who invented it? By asking a person who understand it?
- How do we even determine the prerequisite of a concept? Several math books have their chapter 0/1 talking about basic set theory. Is that necessary?
- If we write the idea using a programming language, then we can just ‘go to definition’ in the IDE!

Why are we encoding spaces into types?

- It brings better extensionality to the type system like bisimulation and function’s extensionality
- It provides a canonical way to represent quotients without breaking subject reduction (as in Lean) or introducing setoid hell (as in Coq)
- It allows us to transport proofs from isomorphic types (by univalence)
- It provides a low-level language to reason over continuous spaces/functions
- It opens the door towards ∞-categories in types

What can it do to a normal programmer?

—

- We can help real-world programming with more expressive type systems (if we open an unsafe mode)
- We can make sure some contracts are satisfied at compile time, and erase the checks or assertions at compile time
- Rust’s lifetime, generalized

Q & A