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.

Mixing types and values

- Lambda cube: an abstraction over the mixture of types and values.
- CIC and MLTT – where types and values are mixed altogether.

In some (old) research PLs, we can mix values into types (similar to constexpr in C++, but in this case, the entire language is constexpr).

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?

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.

Modeling stuffs

How can we exploit the power of DT?

What are types, precisely?

—

- We can see types as sets, and their instances as the elements of sets

Types	Sets
`Nothing`, `!`	∅
Types that talks about values	Families of sets
Functions	Maps of values
Classes, Records, Tuples	Products of sets
Subtypes	Subsets
Equality of values	Equality of elements
…	…

Sets = types, *relations* = ?

- Fun fact: we also encode relations and logical propositions as types

Types	Propositions
`Nothing`, `!`	$\bot$
Types that talks about values	$\forall x, f(x). y$
Functions	$\forall x. y$
Classes, Records, Tuples	$\wedge$
Subtypes	$\vee$
The MLTT Id type	Equality of terms
…	…

Example: 2 is not a rational (proof omitted).

```
theorem : (m : Nat) -> (n : Nat)
  -> 2 * m * m = n * n -> m = 0
```

Corresponding proposition:

$$\forall m, n \in \mathbb{N} \to 2 \times m^2 = n^2 \to m = 0$$

What else can we do?

What if the sets are no longer discrete, but instead continuous?

Can we talk about continuous (preserving topology) functions?

What else can we encode with types?

—

- We can also encode topological spaces as types, and we interpret paths the same way as an ‘equality relation’
- This allows us to talk about spaces and continuous functions in type systems

Types	Spaces
Elements	Points
Equality of values	Paths on points
Functions	Continuous maps of values
…	…

Fundamental groups

The definition of fundamental group can be encoded as a type (that talks about another type):

```
Loop : (A : Type) -> (a : A) -> Type
Loop A a => a = a
```

Then we can prove it to be a group (easy!).

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**!

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 of spheres:

```
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) } )
```

Why types?

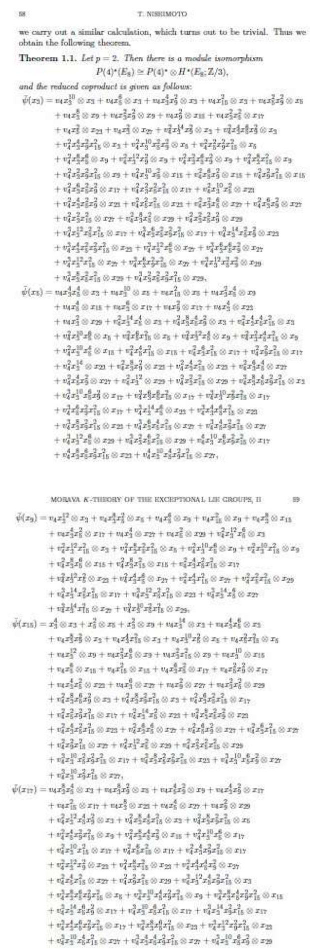
So – what’s the point of all of these?

And how complicated it is?

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

My secret, evil plan

Not going to be a part of the slides ☺

Thank you for your attention

—

jetbrains.com