

# Recent development of dependent types

Slides can't be found in  
<https://github.com/zju-lambda/slides>

Presenter: 千里并封



话说并并,



Assumes:

Dependent Types,  
Curry-Howard Isomorphism  
and  
Generalized Algebraic Data Types



这该死的群友  
数理基础竟  
如此扎实

**Because:**



zju不是人人都爬过lambda cube  
吗。。

# Part I – Vanilla dependent types





- Types depend on values
  - C/C++, Scala
  - Construct types with values as parameters
  - Simple example: C-style sized arrays
- Define by existence

# More?

- Pattern matching
- Dependent functions
  - Telescopes
  - Inaccessible patterns
- Proof by uniqueness
  - Dependent pattern matching
  - GADT eliminators

# Goodies?

- Types are first-class
  - RankN? No need!
  - forall quantification? No need!
  - Type families? No need!
  - Generalize type signatures!
- Type-directed development (meta var)
  - Typed holes
  - Proof search
- Dynamic typing





# Limitations?

- Termination
  - Well-Founded induction or Sized Types
  - Structural induction
- Positivity
  - Strictly positive (``ftw wtf wtf``)
- Slow type-checking
  - If Prelude is changed, all files will be rechecked
- Mutual recursion: lose some definitional equality

```
data P where
  MkP :: (P -> *) -> P

wtf :: P
wtf = MkP (\x -> ftw x x)

ftw :: P -> P -> P
ftw (MkP f) x = f x
```

# Part II – Chocola pattern matching



# Part II – Pattern matching



# Difference?

- “Dependent” pattern matching
  - Impossible patterns
  - Inaccessible patterns (dot? No need!)

```
lookup :: Vec A n -> (m :: Nat)
        -> (m < n) -> A
lookup [ ] _ impossible
lookup (a:_) .Zero      ZNeN          = a
lookup (_:a) .(Suc n) (NNeM _ p) =
    lookup a n p
```

- Consistency
  - Catch-all clauses
  - Overlapping patterns

```
(+) :: Nat -> Nat -> Nat
Zero  + n = n
Suc n + m = Suc $ n + m
n + Zero = n
n + Suc m = Suc $ n + m
```

# Implementations?

- Eliminators
  - Coming from GADT definition
  - Get totality for free
- Case trees
  - Lose catch-all definitional equality for free
  - Friendly for beta-reduction with currying

```
max :: Nat -> Nat -> Nat
max m Zero = m
max Zero m = m
max (Suc n) (Suc m) = Suc $ max n m
```

```
max = \case
  Zero -> id
  Suc n -> \case
    Zero -> Suc n
    Suc m -> Suc $ max n m
```

```
f :: (n :: Nat) -> max n Zero ≡ n
f n = ????
```

# Part III – Sugar? Yes please!



# Primitives?

- $\top/\perp$  types, type universe
- Pi types:  $(\Pi (a : A). F a)$  or  $((a : A) \rightarrow F a)$
- Dependent sum:  $(\Sigma (a : A). B a)$  or  $((a : A) \times B a)$
- Case trees
- Indexed inductive families
- Meta variables
- Typed Lambda Calculus: abstraction, application,  $\alpha \beta \eta$

# Sugars?

- Records
- Pattern matchings
- Implicit arguments
- With-abstractions
- Rewritings



# Type-checking primitives?

- J rule: 
$$\frac{a\ b : A \quad f : \forall a\ b : A. a \equiv b \rightarrow T \quad p : a \equiv b}{f\ p : T}$$



【<|>】Narc

相继式只有你懂了之后才能看懂

- J rule 人话: 当  $a \equiv b$  是 refl 时, context 中的  $b$  都可以变成  $a$
- K rule 人话:  $a \equiv a$  一定是 refl
- 7494 dependent pattern matching's unification

# Do they hold?

- If you postulate  $ua : \forall A B : U \rightarrow A \approx B \rightarrow A \equiv B$ , you can't use K
- $A \approx B$  means "there exists  $f : A \rightarrow B$  and  $g : B \rightarrow A$  and  $f \cdot g = g \cdot f = id$ "
- Agda provides the ability to disable K during pattern matching
- If you don't, they hold
- Agda also supports Path-based HoTT
- Idris only supports with-K



# Part IV – Coinduction

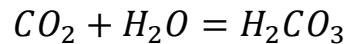
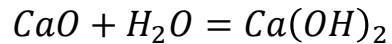
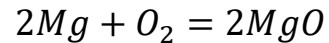
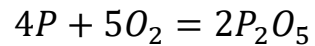
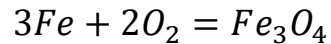
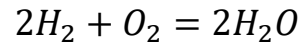


# Algebra?

- What're *Algebraic Data Types*?
  - $1 + A \times \text{List}(A) \rightarrow \text{List}(A)$
- Functor
  - $F(A, X) := 1 + A \times X$  defines a Functor
- Algebra
  - $F(A, \text{List}(A)) \rightarrow \text{List}(A)$  is an algebra
  - Simplify some trivial arguments:  $F(A) \rightarrow A$

# Algebra!

- Talking about how to construct something



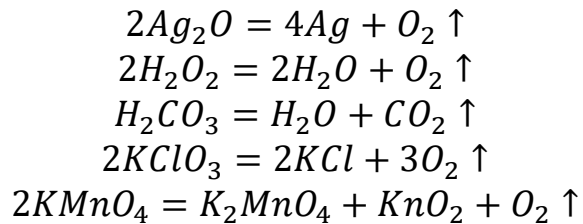
# Coalgebra?

- What're *Coalgebraic Data Types*?
  - $Colist(A) \rightarrow 1 + A \times Colist(A)$
- Coalgebra
  - $Colist(A) \rightarrow F(A, Colist(A))$  is a coalgebra
  - Simplify some trivial arguments:  $A \rightarrow F(A)$
  - Notice the only difference is: the arrow is flipped

# Coalgebra!

- Talking about how to destruct something
- Copatterns

```
ones :: Stream Nat  
.head ones = Suc Zero  
.tail ones = ones
```



# Guarded recursion?

- Look at this function, infinite loop!

```
ones :: Stream Nat
.head ones = Suc Zero
.tail ones = ones
```

- However, if we lazy the data structure, it can be used finitely
- **Disallowing myself to be further destructed**
- Sounds like the dual of structural induction?
- We call it *coinduction*



# What's wrong?

- We can't reduce coinductive structures (coalgebras) into normal form
- How can we decide if two coinductive structures are equivalent?

# Yes we can!

- Equivalence on coinductive types
  - It's a **relation**
  - Notation:  $\forall a\ b : A \rightarrow a \cong b$
- Bisimulation
  - Recall  $Stream(A) \rightarrow A \times Stream(A)$
  - $head : Stream(A) \rightarrow A$
  - $tail : Stream(A) \rightarrow Stream(A)$
  - $$\frac{a\ b : Stream(A) \quad head\ a \equiv head\ b \quad tail\ a \cong tail\ b}{a \cong b}$$



# Bisimulation?

- It's a coinductively-defined **relation**
- $a \cong b \rightarrow (\text{head } a \equiv \text{head } b) \times (\text{tail } a \cong \text{tail } b)$ 
  - We need to “look into” the structure of the type
  - No way to generalize it as the inductive equivalence does
- Which means we need to define bisimulation for every coalgebras
  - Solution? Let's see.

# CPP?

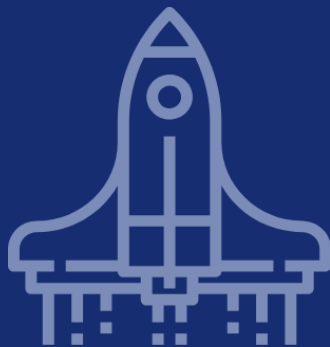
- Coinductive Proof Principle

$\{-\# \text{ LANGUAGE CPP } \#-\}$

$$a \cong b \rightarrow a \equiv b$$

- Why? We'll see.

# Part V – Paths, Faces and Intervals



# Equivalence?

- It's more than an inductively-defined data type
- First let's look at my existing writings (about  $\mathbb{Q}$ ,  $\mathbb{Z}$ ,  $\mathbb{I}$ ):
- <https://ice1000.org/lagda/PathToHigherInductiveTypes.html>
- We now denote Path types  $a \mapsto b$
- Because it's directional
  - dram

# Interval operations?

- Interval has multiple primitive operations
- Let  $p$  be an instance of  $Path\ A\ a\ b$
- Inversion:  $\langle i \rangle p (\sim i)$ , (type:  $Path\ A\ b\ a$ )
- Conjunction/Disjunction:  $\langle i\ j \rangle p (i \vee j)$  and  $\langle i\ j \rangle p (i \wedge j)$
- De Morgan Algebra:  $0 \vee i = i$ ,  $1 \vee i = 1$ ,  $\sim(i \vee j) = \sim i \wedge \sim j$
- If an Interval is in context, then we're in a path
- Paths are complete path, we can't determine which endpoints are you in

# What do we already have?

- Haskell's "." is  $\text{cong} : (f : A \rightarrow B) \rightarrow (x \mapsto y) \rightarrow (f x \mapsto f y)$ 
  - $\text{cong } f \, p = \langle i \rangle f (p \, i)$
- Haskell's "flip" is  $\text{funExt} : (p : (a : A) \rightarrow f \, a \mapsto g \, a) \rightarrow (f \mapsto g)$ 
  - $\text{funExt } p = \langle i \rangle \lambda a \rightarrow p \, a \, i$
- Haskell's "const" is  $\text{refl} : a \rightarrow (a \mapsto a)$ 
  - $\text{refl } a = \langle \_ \rangle a$
- "Composing ~" is  $\text{sym} : (a \mapsto b) \rightarrow (b \mapsto a)$ 
  - $\text{sym } p = \langle i \rangle p (\sim i)$



# Face lattice?

- “Face”,  $\mathbb{F}$ , is a lattice, a sub-polyhedra

$i : \mathbb{I}, (i = 0) \vee (i = 1) \vdash A$	$A(i0) \bullet \quad A(i1) \bullet$
$i : \mathbb{I}, j : \mathbb{I}, (i = 0) \vee (j = 1) \vdash A$	$  \begin{array}{c}  A(i0)(j1) \xrightarrow{A(j1)} A(i1)(j1) \\  \uparrow A(i0) \\  A(i0)(j0)  \end{array}  $
$i : \mathbb{I}, j : \mathbb{I}, (i = 0) \vee (i = 1) \vee (j = 0) \vdash A$	$  \begin{array}{ccc}  A(i0)(j1) & & A(i1)(j1) \\  \uparrow A(i0) & & \uparrow A(i1) \\  A(i0)(j0) & \xrightarrow{A(j0)} & A(i1)(j0)  \end{array}  $

# Face lattice?

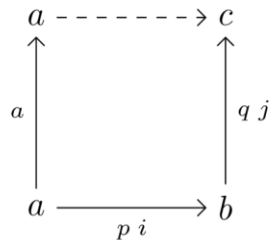
- “Face”,  $\mathbb{F}$ , is a lattice,  $0_{\mathbb{F}}, 1_{\mathbb{F}}, (i = 0), (i = 1), \vee, \wedge$
- $\psi : \mathbb{F}$  are union of faces
- If a face is in context, then we’re in a sub-polyhedra

# Systems?

- Systems are a list of  $\psi \mapsto a$ , denoted as  $[\psi_0 \mapsto a_0, \psi_1 \mapsto a_1, \dots, \psi_n \mapsto a_n]$
- Each  $a_i$  is a “Partial” mappings on Intervals
- Each  $\psi_i$  is an “extent”
- $\sum \psi_i = 1_{\mathbb{F}}$

# Composition?

- The blog didn't cover "transitivity"
  - $(a \mapsto b) \rightarrow (b \mapsto c) \rightarrow (a \mapsto c)$
- It's done by the "comp" operation (CCHM)
- "comp": if a partial path is *extendible* (?) at  $0_{\mathbb{F}}$ , it is at  $1_{\mathbb{F}}$  too
- It's used to fill a partial square/cube/tesseract
- $p : a \mapsto b, q : b \mapsto c$
- $\langle i \rangle \text{ comp refl } (p \ i) \ [i = 0_{\mathbb{F}} \rightarrow \langle j \rangle a, i = 1_{\mathbb{F}} \rightarrow \langle j \rangle q \ j]$
- Simplify:  $\langle i \rangle \text{ comp } (p \ i) \ [i = 0 \rightarrow \text{refl}, i = 1 \rightarrow q]$



# Transport?

- “comp”: if a partial path is *extensible* (?) at  $0_{\mathbb{F}}$ , it is at  $1_{\mathbb{F}}$  too
- ???
- $\forall a : A, p : A \mapsto B. \text{comp } p \ a \ [\ ] \equiv b : B$
- So we define  $\text{transport } p \ a = \text{comp } p \ a \ [\ ]$
- Coercion for heterogenous equality

# Glue?

- Univalence Axiom can be proved via Gluing
- 已经看不懂了



没看model, 看不懂了, pres开始就已经上升到我不太能接受的复杂度了



# Cubical Agda!

- Currently we can play with Cubical Agda
  - New style pattern matching for Systems
  - Tons of built-in definitions, Sub, Partial, \*comp, \*glue
  - Higher Inductive Families
- Highly WIP (developed by Mr. 3 Eyes)
  - Does not compute on inductive families
  - Does not support “with” in path patterns

# Bisimulation is Path!

- Path is all about conversion, no need to normalize
- Bisimulation is Path!
- Interesting Conat: <https://github.com/agda/cubical/pull/57/files>
- Proved that bisimulation on Conat is equivalence, bisimulation is also a proposition (hProp)



# Absurd?

- How do we do absurd patterns (eliminate impossible stuffs)?
- Because we no longer can pattern match on equality types
- Imagine we have a  $p : \text{zero} \mapsto \text{suc } n$
- Create function  $f : \text{zero} = \perp; f : \text{suc } n = \top$
- $p_1 = \text{cong } f \ p$  has type  $\perp \mapsto \top$
- $\text{transport } p_1 \ tt$  is an instance of  $\perp$
- Now we have a pattern-matchable thing

# Part VI – OwO



# OwO?

- Programming Language
  - Consistent Coinduction/Induction (Agda sucks here)
  - Path, Interval, but friendlier for programming
  - Proof search, catch-all support
- 
- Linear/Affine type system? (Rust backend?)
  - Coinductive IO?
  - Using cubicaltt as core language?



# QAQ?

- Agda
  - Legacies
  - Poorly maintained Agsy/JS backend/Case-split/Emacs mode
  - Unnecessary compiler components are coupled
- Idris
  - Reduction bug
  - Too few features
  - Everyone's heading to Blodwen (QTT!)
- Coq
  - Far from a programming language
  - Unreadable without IDE



【<+>】Narc

五年没人动都比 idris 强

# =\_=#?

- Haskell
  - Too weak
  - Purpose
- Scala
  - Purpose
  - OO
- C++
  - ???



钱泽森 @riaqn\_zh · 1 月 31 日

说实话 Haskell 这么贫瘠的类型系统能玩出这么多花来 (Monad, STM, ForkIO) 也是听不容易的。

[翻译推文](#)



1



1



# QwQ!



咖啡忘了加糖

先读论文吧 骚年 全村的希望啊 我已经半放弃了

# Questions?

