Introduction to dependent types

Presented by Tesla Ice Zhang (ice1000)
Sponsored by ZJU-Lambda

Slides can be found in
https://github.com/zju-lambda/slides

Part I – What has already happened

Step zero

- In the beginning, there was nothing
  - Imagine we have a BCPL (or, JavaScript)
  - The values are not typed, and varies via implicit conversions
  - The Spirit of God moves upon the bash shell
- God says, “there should be types.”
  - Imagine we have a Golang
- And there was types
- And God saw the types, that It was good: and God divided types from values

Step zero

- And God saw the types, that It was good: and God divided types from values
- And there're functions, which converts values from values
- Functions are equipped with types, type checked at compile time
- And the compile time and runtime, types and values.
- This is the 0th step

```
public class Marisa {
    public static void main(String ... args) {
        Marisa marisa = new Marisa();
    }
}
```

Step (suc zero)

- And God said, let's have a family of types
- And there're type constructors, which are types parameterized by types

```java
public class Marisa<T> {
    public static void main(String ... args) {
        Marisa<Marisa<?>> marisa = new Marisa<>();
    }
}
```

```haskell
module Marisa where

data Tree a
  = Leaf a
  | Node a (Tree a) (Tree a)
```

PingCAP

Step (suc zero)

- And God said, “values should be grouped with types”
- And there’re algebraic data types

```
module Marisa where

data SpellCards
  = MasterSpark
  | FinalSpark
  | UndirectionalLaser
```

PingCAP

Step (suc zero)

- And God said, “values should be constructed with other values via constructors”
- And there’re parameterized data constructors

```
module Marisa where

data Expr
  = IntLit Int
  | BoolLit Bool
  | IfExpr Expr Expr Expr
```

PingCAP

Step (suc zero)

- There're expressions constructed with pure constructors (normal form), or with function applications (**red**ucible **ex**pressions, redex)
- Redex are "reduced" via α-conversions (renaming), β-reductions (cancelling redundant parameters) and η-conversions (inlining) to normal forms at runtime

Step (suc zero)

- Which is a simply-typed lambda calculus equipped with ADTs
- Function parameters are “for all”s, ADTs are “or”s, tuples are “and”s
- Which is also first-order logic
- This is the 1st step

{a : Nat} -> Step (suc (suc a))

- And many things happen
- We have generics, generalizing functions over type variables
- We have GADTs, where the return type of the constructors can be specified
- We have Type Families, which are functions over types
- We have Type Classes, which is ad-hoc polymorphism
- We have Haskell

{a : Nat} -> Step (suc (suc a))

- And many things happen
- We have generics, generalizing functions over type variables
- We have GADTs, where the return type of the constructors can be specified
- We have Type Families, which are functions over types
- We have Type Classes, which is ad-hoc polymorphism

{a : Nat} -> Step (suc (suc a))

- Wrong design
- Type parameters are always inferred, value parameters are never inferred
- We have GADTs, where the return type of the constructors can be specified
- We have Type Families, which are functions over types
- We have Type Classes, which is ad-hoc polymorphism

PingCAP

{a : Nat} -> Step (suc (suc a))

- Wrong design
- Type parameters are always inferred, value parameters are never inferred
- Type parameters in ADT are all solid, in GADT are all varying
- We have Type Families, which are functions over types
- We have Type Classes, which is ad-hoc polymorphism

PingCAP

{a : Nat} -> Step (suc (suc a))

- Wrong design
- Type parameters are always inferred, value parameters are never inferred
- Type parameters in ADT are all solid, in GADT are all varying
- Type Families are just arbitrary functions
- We have Type Classes, which is ad-hoc polymorphism

PingCAP

{a : Nat} -> Step (suc (suc a))

- Wrong design
- Type parameters are always inferred, value parameters are never inferred
- Type parameters in ADT are all solid, in GADT are all varying
- Type Families are just arbitrary functions
- Type Classes are just implicit module argument

{a : Nat} -> Step (suc (suc a))

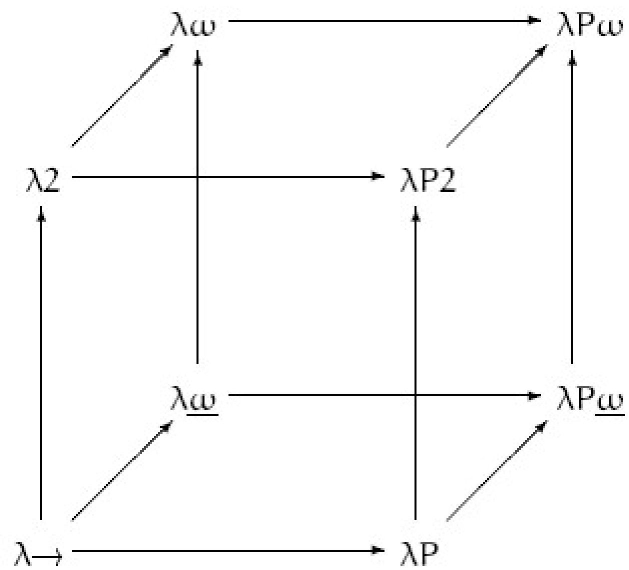
- Wrong design
- Type parameters are always inferred, value parameters are never inferred
- Type parameters in ADT are all solid, in GADT are all varying
- Type Families are just arbitrary functions
- Type Classes are just implicit module argument
- 无駄无駄无駄无駄无駄无駄无駄无駄无駄

{a : Nat} -> Step (suc (suc a))

- Type parameters are always inferred, value parameters are never inferred
- Type parameters in ADT are all solid, in GADT are all varying
- Type Families are just arbitrary functions
- Type Classes are just implicit module argument

Let’s say we want to build a Programming Language

- From the beginning
- What’s wrong with the plain old PLs?
  - Types are all constants (λ2)
  - Sometimes parameterized (λΠ)
- Take a look at the lambda cube
- We need λω



PingCAP

Part II – How can we solve that

Correct functions generalized over types

- In Haskell, functions are limited
  - All type parameters are **implicit**
  - All value parameters **explicit**
- They should be customizable
- Or, to make Mr. Long happy, "***HACKABLE***"

```
-- | `id 233` returns `233`
id :: {A :: Type} → (a :: A) → A
id {A} a = a

-- | `id' Int 233` returns `233`
id' :: (A :: Type) → (a :: A) → A
id' A a = a

data Unit = unit
-- | `id'' Unit` returns `unit`
id'' :: (A :: Type) → {a :: A} → A
id'' A {a} = a
```

PingCAP

Correct functions generalized over types

- id 有三种写法，你可知道?

```
-- | `id 233` returns `233`
id :: {A :: Type} → (a :: A) → A
id {A} a = a

-- | `id' Int 233` returns `233`
id' :: (A :: Type) → (a :: A) → A
id' A a = a

data Unit = unit
-- | `id'' Unit` returns `unit`
id'' :: (A :: Type) → {a :: A} → A
id'' A {a} = a
```

Correct functions generalized over types

- Implicit arguments are automatically inferred and passed by the compiler, explicit arguments are passed by users
- Un-inferred implicit arguments are called “meta variables”
- It’s a long process, to get really get resolved
- It’s an intense feeling for the other type checkers

```
-- | `id 233` returns `233`
id :: {A :: Type} → (a :: A) → A
id {A} a = a

-- | `id' Int 233` returns `233`
id' :: (A :: Type) → (a :: A) → A
id' A a = a

data Unit = unit
-- | `id'' Unit` returns `unit`
id'' :: (A :: Type) → {a :: A} → A
id'' A {a} = a
```

PingCAP

Correct GADTs

- In Haskell, GADTs are limited
  - All type parameters are **indices**
  - In ADTs they're all **parameter**
- They should be clarified

```
data Equality {A :: Type} (a :: A)
    :: A → Type where
  reflexive :: Equality a a
```

PingCAP

Correct GADTs

- The part “Eq {A : Type} (a :: A)” is a plain old ADT’s type part
- The “A” in “A -> Type” is a plain old GADT’s type parameter part

```
data Equality {A :: Type} (a :: A)
    :: A → Type where
  reflexive :: Equality a a
```

PingCAP

Correct GADTs

- With Haskell's weak GADT we can only have

```
data Equality' :: {A :: Type}
    -> A -> A -> Type where
  reflexive' :: {A :: Type} -> {a :: A}
    -> Equality' {A} a a
```

PingCAP

Correct GADTs

- With Haskell's weak GADT we can only have
- Some boilerplate codes

```
data Equality' :: {A :: Type}
    → A → A → Type where
  reflexive' :: {A :: Type} → {a :: A}
    → Equality' {A} a a
```

PingCAP

Correct GADTs

- With Haskell's weak GADT we can only have
- Some boilerplate codes
- But Haskell **does not support** specifying whether the parameters are implicit or not, so it's not that boilerplate

PingCAP

Correct GADTs

- With Haskell's weak GADT we can only have
- Some boilerplate codes
- But Haskell **does not support** specifying whether the parameters are implicit or not, so it's not that boilerplate

Correct Type Families

- That’s it
- Why bother?

```
data Nat = Zero | Suc Nat
(+) :: Nat → Nat → Nat
Zero + n = n
(Suc m) + n = Suc (m + n)
```

Part III – What's next

A serious problem in correct Type Families

- That's it
- Why bother?
- Haskell: because you'll need to "run" this function at compile time to type check functions with types like

```
(a b :: Nat) →
  Equality (b + a) (a + b)
```

```
data Nat = Zero / Suc Nat
(+) :: Nat → Nat → Nat
Zero + n = n
(Suc m) + n = Suc (m + n)
```

PingCAP

Reduction

- Then we reduce redexes at compile time
- So we can have

```
reflexive :: Equality
 (Suc Zero + Zero)
 (Suc Zero)
```

- That’s so easy

PingCAP

Reduction

- What if the functions we need to reduce are:
  - Not terminating
  - Pattern-match on its parameters but not covering all input cases

PingCAP

Reduction

- What if the functions we need to reduce are:
  - Not terminating
  - Pattern-match on its parameters but not covering all input cases
- We need to ensure all the functions we're reducing are **total**
- We can have functions only reduces at runtime, non-terminating/panicking

PingCAP

Turing-incompleteness

- Termination means Turing-incompleteness

Turing-incompleteness

- Termination means Turing-incompleteness

PingCAP

Context-splitting

- It’s about complex functions which requires you to pattern-match on its parameters

PingCAP

Context-splitting

- It’s about complex functions which requires you to pattern-match on its parameters
- In this kind of pattern matching, the context changes according to patterns due to the function indices

PingCAP

Context-splitting

- Like, if `Equality a b` is matched to `reflexive`
- According to the return type of `reflexive` is `Equality a a`
- `b` must be `a`

Context-splitting

- Like, if `Equality a b` is matched to `reflexive`
- According to the return type of `reflexive` is `Equality a a`
- `b` must be `a`
- Pattern matching brings information to the context

PingCAP

Context-splitting

- Pattern matchings in Haskell are all "positively succeeding"
- Like, a parameter can be matched by some constructors

```
map :: (a → b) → [a] → [b]
map _ [] = []
map f (a : as) → f a : map f as
```

PingCAP

Context-splitting

- If the indices of a type never present in any data constructors, pattern matching “negatively succeeds”

PingCAP

Context-splitting

- If the indices of a type never present in any data constructors, pattern matching “negatively succeeds”
- We use a special pattern to indicate that this pattern matching “negatively succeeds”

```
wtf :: Equality Zero (Suc Zero) → Unit
wtf impossible
```

Context-splitting

- If 没有这种操作 pattern-matching, we reach a “failed” result
- Three results of pattern matching

Context-splitting

- Sometimes one pattern matching makes another pattern obvious
- The obvious pattern becomes meaningless, we call it “inaccessible patterns”, in Agda “dot patterns”

Context-splitting

- These interesting pattern-matchings are called

Dependent pattern matching

Positivity

- Functions should always terminate
- Functions pattern-match on parameters

PingCAP

Positivity

- Imagine we have an ADT with a function on it like this

```
data NonPos :: Type where
  Evil :: (NonPos → NonPos)
       → NonPos

noNoNo :: NonPos → NonPos → NonPos
noNoNo (Evil f) x = f x
```

PingCAP

Positivity

- Imagine we have an ADT with a function on it like this
- `noNoNo x x` returns `noNoNo x x`
- No way to reduce this, no normal form
- You bad bad

```
data NonPos :: Type where
  Evil :: (NonPos → NonPos)
       → NonPos

noNoNo :: NonPos → NonPos → NonPos
noNoNo (Evil f) x = f x
```

PingCAP

Higher Inductive Types

- What's wrong with simple GADTs
- Agda's `Data.Int`

```
data Int : Set where
  pos    : (n : Nat) → Int
  negsuc : (n : Nat) → Int
```

```
open import Agda.Builtin.Int public
  using ()
  renaming
  ( Int to ℤ
  ; pos    to +_       -- "+ n"      stands for "n"
  ; negsuc to -[1+_]  -- "-[1+ n ]" stands for "- (1 + n)"
  )
```

PingCAP

Higher Inductive Types

- Proving plus-associative law

```
+-assoc : Associative _+_
+-assoc (+ zero) y z rewrite +-identityˡ       y   | +-identityˡ (y + z) = refl
+-assoc x (+ zero) z rewrite +-identityʳ  x        | +-identityˡ      z  = refl
+-assoc x y (+ zero) rewrite +-identityʳ (x + y) | +-identityʳ  y       = refl
+-assoc -[1+ a ]  -[1+ b ]  (+ suc c) = sym (distribʳ-⊖-+-neg a c b)
+-assoc -[1+ a ]  (+ suc b) (+ suc c) = distribˡ-⊖-+-pos (suc c) b a
+-assoc (+ suc a) -[1+ b ]  -[1+ c ]  = distribˡ-⊖-+-neg c a b
+-assoc (+ suc a) -[1+ b ] (+ suc c)
  rewrite distribˡ-⊖-+-pos (suc c) a b
        | distribʳ-⊖-+-pos (suc a) c b
        | sym (ℕₚ.+-assoc a 1 c)
        | ℕₚ.+-comm a 1
        = refl
+-assoc (+ suc a) (+ suc b) -[1+ c ]
  rewrite distribʳ-⊖-+-pos (suc a) b c
        | sym (ℕₚ.+-assoc a 1 b)
        | ℕₚ.+-comm a 1
        = refl
+-assoc -[1+ a ] -[1+ b ] -[1+ c ]
  rewrite sym (ℕₚ.+-assoc a 1 (b ℕ.+ c))
        | ℕₚ.+-comm a 1
        | ℕₚ.+-assoc a b c
        = refl
+-assoc -[1+ a ] (+ suc b) -[1+ c ]
  rewrite distribʳ-⊖-+-neg a b c
        | distribˡ-⊖-+-neg c b a
        = refl
+-assoc (+ suc a) (+ suc b) (+ suc c)
  rewrite ℕₚ.+-assoc (suc a) (suc b) (suc c)
        = refl
```

Higher Inductive Types

- Proving plus-associative law (written by me)

```
a+/b+c/=/a+b/+c : ∀ a b c → a + (b + c) ≡ a + b + c
a+/b+c/=/a+b/+c (+ zero) b c rewrite 0+a=a (b + c) | 0+a=a b = refl
a+/b+c/=/a+b/+c a (+ zero) c rewrite 0+a=a c | a+0=a a = refl
a+/b+c/=/a+b/+c a b (+ zero) rewrite a+0=a b | a+0=a (a + b) = refl
a+/b+c/=/a+b/+c (+ a) (+ b) (+ c)
  rewrite nat-add-assoc a b c = refl
a+/b+c/=/a+b/+c -[1+ a ] -[1+ b ] (+ nsuc c)
  rewrite -a+/b-c/=b-/a+c/ a c b = refl
a+/b+c/=/a+b/+c -[1+ a ] (+ nsuc b) -[1+ c ]
  rewrite -a+/b-c/=b-/a+c/ a b c
        | b-c-a=b-/c+a/ c b a
        = refl
a+/b+c/=/a+b/+c (+ nsuc a) -[1+ b ] -[1+ c ]
  rewrite b-c-a=b-/c+a/ c a b = refl
a+/b+c/=/a+b/+c (+ nsuc a) -[1+ b ] (+ nsuc c)
  rewrite a-b+c=a+c-b a b $ nsuc c
        | a+/b-c/=a+b-c (nsuc a) c b
        | sym $ nat-add-assoc a 1 c
        | nat-add-comm a 1
        = refl
a+/b+c/=/a+b/+c -[1+ a ] (+ nsuc b) (+ nsuc c)
  rewrite a-b+c=a+c-b b a (nsuc c) = refl
a+/b+c/=/a+b/+c -[1+ a ] -[1+ b ] -[1+ c ]
  rewrite nat-add-comm a $ nsuc $ b :+: c
        | nat-add-comm (b :+: c) a
        | nat-add-assoc a b c
        = refl
a+/b+c/=/a+b/+c (+ nsuc a) (+ nsuc b) -[1+ c ]
  rewrite a+/b-c/=a+b-c (nsuc a) b c
        | sym $ nat-add-assoc a 1 b
        | nat-add-comm a 1
        = refl
```

Higher Inductive Types

- How egg pain
- 好 蛋 疼

PingCAP

Higher Inductive Types

- What if we can define equality within GADTs
- Data constructors are “point”s
- Equalities are “line”s

Higher Inductive Types

- What if we can define equality within GADTs
- Data constructors are “point”s
- Paths ~~Equalities~~ are “line”s

```
data Int :: Type where
  pos :: (n :: Nat) → Int
  neg :: (n :: Nat) → Int
  equiv :: pos Zero ≡ neg Zero
```

PingCAP

Higher Inductive Types

- Pattern matching on HITs is related to the implementation of Paths
- Out of topic, maybe next time

```
data Int :: Type where
  pos :: (n :: Nat) → Int
  neg :: (n :: Nat) → Int
  equiv :: pos Zero ≡ neg Zero
```

PingCAP

Agda’s HIT

- Agda has an experimental HIT implementation
- Path, Id, etc.
- refl becomes a function

Agda’s HIT

- HIT Int definition, with succ, pred
- Depends on Nat

```
open import Agda.Builtin.Cubical.Path
open import Agda.Primitive.Cubical

data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data ℤ : Set where
  | pos : (n : ℕ) → ℤ
  | neg : (n : ℕ) → ℤ
  | zeroEq : pos zero ≡ neg zero
  |
sucℤ : ℤ → ℤ
sucℤ (pos n) = pos (suc n)
sucℤ (neg zero) = pos (suc zero)
sucℤ (neg (suc n)) = neg n
sucℤ (zeroEq x) = pos (suc zero)

preℤ : ℤ → ℤ
preℤ (pos zero) = neg (suc zero)
preℤ (pos (suc n)) = pos n
preℤ (neg n) = neg (suc n)
preℤ (zeroEq x) = neg (suc zero)
```

Agda’s HIT

- Plus for HIT Int
- Depends on succ, pred

- Also +, * for Nat by the way

```
_+_ : ℤ → ℤ → ℤ
pos zero + b = b
pos (suc n) + b = sucℤ (pos n + b)
neg zero + b = b
neg (suc n) + b = preℤ (neg n + b)
zeroEq _ + b = b
```

```
_:+:_ : ℕ → ℕ → ℕ
zero :+: b = b
suc a :+: b = suc (a :+: b)

_×_ : ℕ → ℕ → ℕ
zero × _ = zero
suc a × b = b :+: (a × b)
```

Agda's HIT

- Simple check for correctness

```
testInt : pos zero ≡ neg zero
testInt = zeroEq

testInt′ : preℤ (pos (suc zero)) ≡ neg zero
testInt′ = zeroEq
```

PingCAP

Agda’s HIT

- Using `Id` type for equality proof
- refl is a function based on `Id`

- I don’t know much about the tech details behind

```
refl : ∀ {ℓ} {A : Set ℓ} (x : A) → Id x x
refl x = conid i1 (λ _ → x)

_==_ = Id

testInt′′ : preℤ (pos zero) == neg (suc zero)
testInt′′ = refl (neg (suc zero))
```

PingCAP

Agda’s HIT

- We can have dividing
- Without postulating that “a/b = (a*c)/(b*c)”
- Because that’s just a path

```
data ℚ : Set where
  _÷_ : (a b : ℕ) → ℚ
  divEq : (a b c : ℕ)
    → a ÷ b ≡ (a × c) ÷ (b × c)

one = suc zero
two = suc one
four = suc (suc two)

testQuo : (two ÷ one) ≡ (four ÷ two)
testQuo = divEq two one two
```

Agda’s HIT

- Very much WIP
- Cannot even pattern match on two HITs at the same time

PingCAP

Part IV – Type Theory

Basics

- Unit value
- Empty value
- Dependent Product
- Dependent Coproduct
- Pi Types
- Eliminators

PingCAP

Basics

- Unit value (True)
- Empty value (False)
- Dependent Product (Dependent Records)
- Dependent Coproduct (GADTs)
- Pi Types (Functions with Telescopes)
- Eliminators (Pattern matching)

PingCAP

Basics

- Unit value (True)
- Empty value (False)
- Dependent Product (Dependent Records)
- Dependent Coproduct (GADTs)
- Pi Types (Functions with Telescopes)
- Eliminators (Pattern matching)

$$
\begin{array}{llll}
s,\ t,\ A,\ B & ::= & x & \text{variable} \\
& \mid & (x : A) \to B & \text{dependent function type} \\
& \mid & \lambda x.\ t & \text{lambda abstraction} \\
& \mid & s\ t & \text{function application} \\
& \mid & (x : A) \times B & \text{dependent pair type} \\
& \mid & \langle s,\ t \rangle & \text{dependent pairs} \\
& \mid & \pi_1\ t \ \mid\ \pi_2\ t & \text{projection} \\
& \mid & \mathsf{Set}_i & \text{universes } (i \in \{0..\}) \\
& \mid & 1 & \text{the unit type} \\
& \mid & \langle\rangle & \text{the element of the unit type} \\
\Gamma,\ \Delta & ::= & \varepsilon & \\
& \mid & (x : A)\Gamma & \text{telescopes}
\end{array}
$$

PingCAP

Type Checking

Contexts: $\boxed{\Gamma \vdash \textbf{valid}}$

$$\frac{}{\vdash \textbf{valid}} \qquad \frac{\Gamma \vdash \textbf{valid} \qquad \Gamma \vdash A : \mathsf{Set}_i}{\Gamma, x : A \vdash \textbf{valid}}$$

Types and terms: $\boxed{\Gamma \vdash t : A}$

$$\frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1}} \qquad \frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x : A \vdash B : \mathsf{Set}_i}{\Gamma \vdash (x : A) \times B : \mathsf{Set}_i}$$

$$\frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x : A \vdash B : \mathsf{Set}_i}{\Gamma \vdash (x : A) \to B : \mathsf{Set}_i} \qquad \frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash 1 : \mathsf{Set}_0} \qquad \frac{\Gamma \vdash \textbf{valid} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B[x := s]}{\Gamma \vdash \langle s, t \rangle : (x : A) \times B} \qquad \frac{\Gamma \vdash t : (x : A) \times B}{\Gamma \vdash \pi_1\ t : A} \qquad \frac{\Gamma \vdash t : (x : A) \times B}{\Gamma \vdash \pi_2\ t : B[x := \pi_1\ t]}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x. t : (x : A) \to B} \qquad \frac{\Gamma \vdash s : (x : A) \to B \qquad \Gamma \vdash t : A}{\Gamma \vdash s\ t : B[x := t]} \qquad \frac{\Gamma \vdash \textbf{valid}}{\Gamma \vdash \langle\rangle : 1}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A \leqslant B}{\Gamma \vdash t : B}$$

PingCAP

Useful syntactic sugars

- GADTs
- Dependent Records
- Dependent Pattern Matching
- With-Abstraction
- Rewriting
- Projection functions (destructors)

Part V – Useful things

Views

Universes

Prop/Proof Irrelevance

Decidables

Well-Founded Inductions

Auto-Generalize

Coinduction/Copatterns

Reflection/Tactics

Compilation: Erasure or Type-Preserving

PingCAP

PingCAP

Thank You !