Recent development of dependent types

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

PingCAP

Presenter: 千里冰封

Assumes:

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

PingCAP

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
```

PingCAP

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
```

PingCAP

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?

- $\top/\bot$ types, type universe
- Pi types: $(\Pi\ (a : A). F\ a)$ or $\left((a : A) \to F\ a\right)$
- Dependent sum: $(\Sigma\ (a : A). B\ a)$ or $\left((a : A) \times B\ a\right)$
- Case trees
- Indexed inductive families
- Meta variables
- Typed Lambda Calculus: abstraction, application, α β η

Sugars?

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

Type-checking primitives?

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

- 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

PingCAP

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

PingCAP

Part IV – Coinduction

Algebra?

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

Algebra!

- Talking about how to construct something

$$2H_2 + O_2 = 2H_2O$$
$$3Fe + 2O_2 = Fe_3O_4$$
$$4P + 5O_2 = 2P_2O_5$$
$$2Mg + O_2 = 2MgO$$
$$CaO + H_2O = Ca(OH)_2$$
$$CO_2 + H_2O = H_2CO_3$$

PingCAP

Coalgebra?

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

Coalgebra!

- Talking about how to destruct something

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

- Copatterns

$$
\begin{gathered}
2Ag_2O = 4Ag + O_2 \uparrow \\
2H_2O_2 = 2H_2O + O_2 \uparrow \\
H_2CO_3 = H_2O + CO_2 \uparrow \\
2KClO_3 = 2KCl + 3O_2 \uparrow \\
2KMnO_4 = K_2MnO_4 + KnO_2 + O_2 \uparrow
\end{gathered}
$$

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: $\forall\, a\ b : A \to a \cong b$
- Bisimulation
  - Recall $Stream(A) \to A \times Stream(A)$
  - $head : Stream(A) \to A$
  - tail : $Stream(A) \to Stream(A)$
  - $\dfrac{a\ b : Stream(A) \quad head\ a \cong head\ b \quad tail\ a \cong tail\ b}{a \cong b}$

PingCAP

Bisimulation?

- It's a coinductively-defined **relation**
- $a \cong b \rightarrow (head\ a \equiv head\ b) \times (tail\ a \cong tail\ b)$
  - 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

```
{-# LANGUAGE CPP #-}
```

$$a \cong b \to a \equiv b$$

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

Face lattice?

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

$i : \mathbb{I}, (i = 0) \vee (i = 1) \vdash A$	$A(i0) \bullet \qquad A(i1) \bullet$
$i : \mathbb{I}, j : \mathbb{I}, (i = 0) \vee (j = 1) \vdash A$	$\begin{tikzcd} A(i0)(j1) \arrow[r, "A(j1)"] & A(i1)(j1) \\ A(i0)(j0) \arrow[u, "A(i0)"] & \end{tikzcd}$
$i : \mathbb{I}, j : \mathbb{I}, (i = 0) \vee (i = 1) \vee (j = 0) \vdash A$	$\begin{tikzcd} A(i0)(j1) & A(i1)(j1) \\ A(i0)(j0) \arrow[u, "A(i0)"] \arrow[r, "A(j0)"'] & A(i1)(j0) \arrow[u, "A(i1)"'] \end{tikzcd}$

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 $[\psi_0 \mapsto a_0, \psi_1 \mapsto a_1, \ldots, \psi_n \mapsto a_n]$

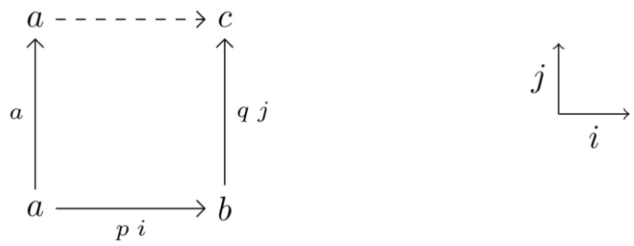
- Each a_i is a “Partial” mappings on Intervals
- Each ψ_i is an “extent”
- $\sum \psi_i = 1_{\mathbb{F}}$

Composition?

- The blog didn't cover "transitivity"
  - $(a \mapsto b) \to (b \mapsto c) \to (a \mapsto c)$

- It's done by the "comp" operation (CCHM)
- "comp": if a partial path is *extensible* (?) at $0_{\mathbb{F}}$, it is at $1_{\mathbb{F}}$ too
- It's used to fill a partial square/cube/tesseract

- $p : a \mapsto b, q : b \mapsto c$
- $\langle i \rangle$ comp refl $(p\ i)$ $[i = 0_{\mathbb{F}} \to \langle j \rangle\ a, i = 1_{\mathbb{F}} \to \langle j \rangle\ q\ j]$
- Simplify: $\langle i \rangle$ comp $(p\ i)$ $[i = 0 \to \text{refl}, i = 1 \to q]$



Transport?

- "comp": if a partial path is *extensible* (?) at $0_{\mathbb{F}}$, it is at $1_{\mathbb{F}}$ too

- ???
- $\forall a : A, p : A \mapsto B.\, \mathrm{comp}\; p\; a\; [\,] \equiv b : B$
- So we define $\mathrm{transport}\; p\; a = \mathrm{comp}\; p\; a\; [\,]$

- Coercion for heterogenous equality

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

PingCAP

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 $\mathrm{p} : zero \mapsto suc\ n$
- Create function $f : \mathrm{zero} = \bot; f : \mathrm{suc}\ n = \top$
- $p_1 = \mathrm{cong}\ f\ p$ has type $\bot \mapsto \top$
- $\mathrm{transport}\ p_1\ tt$ is an instance of $\bot$

- Now we have a pattern-matchable thing

Questions?