What are dependent types?

Stephan Boyer
@stepchowfun

Agenda for today

1. Lambda cube
2. Curry-Howard isomorphism
3. A taste of Coq

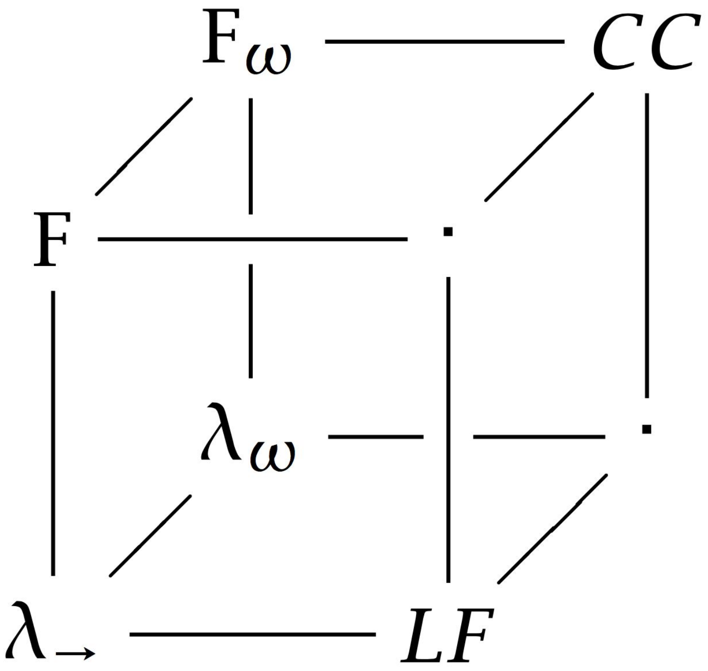

Figure credit: Benjamin C. Pierce, *Types and Programming Languages*

The lambda cube

The simply-typed lambda calculus

Term syntax (t):

- x
- λx : T . t
- t t

Type syntax (T):

- T → T

System F: polymorphism

Term syntax (t):

- x
- $\lambda x : T . t$
- $t\ t$
- $\Lambda X . t$
- $t\ [T]$

Type syntax (T):

- $T \to T$
- X
- $\forall X . T$

System λω: type operators

Term syntax (t):

- x
- $\lambda x : T . t$
- $t\ t$

Type syntax (T):

- $T \rightarrow T$
- X
- $\lambda X : K . T$
- $T\ T$

Kind syntax (K):

- $*$
- $* \rightarrow *$

System LF: dependent types

Term syntax (t):

- x
- $\lambda x : T . t$
- $t\ t$

Type syntax (T):

- $\prod x : T . T$
- $T\ t$

Kind syntax (K):

- $*$
- $\prod x : T . K$

The Curry-Howard isomorphism

Propositions as types

Types:

1. Function type
2. Product type
3. Sum type
4. Dependent product (∏) type
5. Dependent sum (Σ) type
6. Inhabited types
7. The typing rule for application
8. Call/cc
9. ...

Logic:

1. Implication
2. Conjunction
3. Disjunction
4. Universal quantification
5. Existential quantification
6. Provable theorems
7. Modus ponens
8. Peirce’s law
9. ...

A taste of Coq

A taste of Coq

```
Inductive day : Type :=
| monday : day
| tuesday : day
| wednesday : day
| thursday : day
| friday : day
| saturday : day
| sunday : day.
```

```
Definition next_weekday (d : day) : day :=
  match d with
  | monday => tuesday
  | tuesday => wednesday
  | wednesday => thursday
  | thursday => friday
  | friday => monday
  | saturday => monday
  | sunday => monday
  end.
```

A taste of Coq

```
Inductive True : Prop :=
| I : True.

Inductive False : Prop := .

Inductive and (P Q : Prop) : Prop :=
| conj : P -> Q -> (and P Q).
```

A taste of Coq

```
Definition iff (P Q : Prop) := and (P -> Q) (Q -> P).

Inductive or (P Q : Prop) : Prop :=
| or_introl : P -> or P Q
| or_intror : Q -> or P Q.

Definition not (A : Prop) := A -> False.

Inductive eq (X : Type) : X -> X -> Prop :=
| refl_equal : forall x, eq X x x.
```

A taste of Coq

```
Inductive nat : Type :=
| O : nat
| S : nat → nat.

Inductive even : nat -> Prop :=
| even_O : even 0
| even_S : forall n, odd n -> even (S n)
with odd : nat -> Prop :=
  odd_S : forall n, even n -> odd (S n).
```

```
Theorem two_even : even 2.
Proof.
  apply even_S.
  apply odd_S.
  apply even_O.
Qed.
```

A taste of Coq

```
Inductive sig {A : Type} (P : A -> Prop) : Type :=
| exist : forall x : A, P x -> sig P.

Theorem two_even : even 2.
Proof. apply even_S. apply odd_S. apply even_O. Qed.

Definition two : even_nat := exist even 2 two_even.
```

Thanks!

@stepchowfun

References:

1. Benjamin C. Pierce, *Software Foundations*
2. Adam Chlipala, *Certified Programming with Dependent Types*
3. Benjamin C. Pierce, *Types and Programming Languages*
4. Benjamin C. Pierce, *Advanced Topics in Types and Programming Languages*