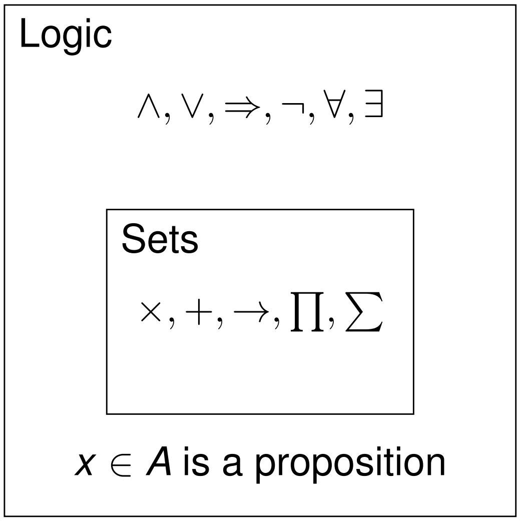
Basics of type theory and Coq

Michael Shulman

January 31, 2012

Type-theoretic foundations

Set theory



Logic

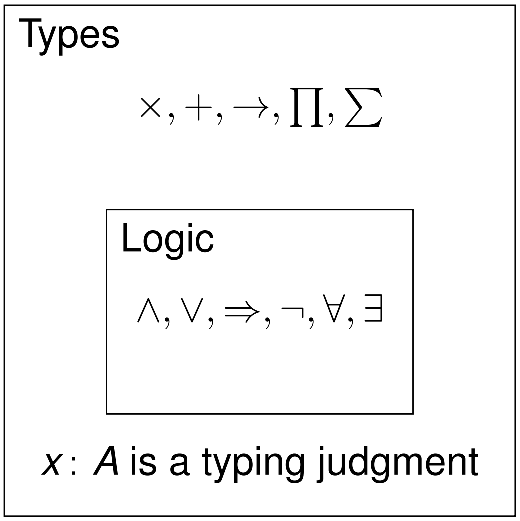
$\wedge, \vee, \Rightarrow, \neg, \forall, \exists$

Sets

$\times, +, \to, \prod, \sum$

$x \in A$ is a proposition

Type theory



Types

$\times, +, \to, \prod, \sum$

Logic

$\wedge, \vee, \Rightarrow, \neg, \forall, \exists$

$x : A$ is a typing judgment

Type theory is programming

For now, think of type theory as a programming language.

- Closely related to functional programming languages like ML, Haskell, Lisp, Scheme.
- More expressive and powerful.
- Can manipulate “mathematical objects”.

Typing judgments

Type theory consists of rules for manipulating judgments.
The most important judgment is a typing judgment:

$$x_1 : A_1,\, x_2 : A_2,\, \ldots x_n : A_n \;\vdash\; b : B$$

The turnstile ⊢ binds most loosely, followed by commas.
This should be read as:

In the context of variables x_1 of type A_1, x_2 of type A_2, . . . , and x_n of type A_n, the expression b has type B.

Examples

$$\begin{aligned} &\vdash\; 0 : \mathbb{N} \\ x : \mathbb{N},\, y : \mathbb{N} &\vdash\; x + y : \mathbb{N} \\ f : \mathbb{R} \to \mathbb{R},\, x : \mathbb{R} &\vdash\; f(x) : \mathbb{R} \\ f : C^\infty(\mathbb{R}, \mathbb{R}),\, n : \mathbb{N} &\vdash\; f^{(n)} : C^\infty(\mathbb{R}, \mathbb{R}) \end{aligned}$$

Type constructors

The basic rules tell us how to construct valid typing judgments, i.e. *how to write programs with given input and output types*. This includes:

1. How to construct new types (judgments $\Gamma \vdash A\colon \text{Type}$).
2. How to construct terms of these types.
3. How to use such terms to construct terms of other types.

Example (Function types)

1. If $A\colon$ Type and $B\colon$ Type, then $A \to B\colon$ Type.
2. If $x\colon A \vdash\ b\colon B$, then $\lambda x^A.b\colon A \to B$.
3. If $a\colon A$ and $f\colon A \to B$, then $f(a)\colon B$.

Derivations

We write these rules as follows.

$$\frac{\vdash A\colon \text{Type} \qquad \vdash B\colon \text{Type}}{\vdash A \to B\colon \text{Type}}$$

$$\frac{x\colon A \vdash b\colon B}{\vdash \lambda x^A.b\colon A \to B}$$

$$\frac{\vdash f\colon A \to B \qquad \vdash a\colon A}{\vdash f(a)\colon B}$$

Derivations in Context

More generally, we allow an arbitrary context $\Gamma = (x_1 : A_1, \ldots, x_n : A_n)$ of typed variables.

$$\frac{\Gamma \vdash A \colon \text{Type} \qquad \Gamma \vdash B \colon \text{Type}}{\Gamma \vdash A \to B \colon \text{Type}}$$

$$\frac{\Gamma, x \colon A \vdash b \colon B}{\Gamma \vdash \lambda x^A . b \colon A \to B}$$

introduction

$$\frac{\Gamma \vdash f \colon A \to B \qquad \Gamma \vdash a \colon A}{\Gamma \vdash f(a) \colon B}$$

elimination

Type theory as programming

This is just a mathematical syntax for programming.

```
int square(int x) { return (x * x); }
```

```
def square(x):
  return (x * x)
```

```
square :: Int -> Int
square x = x * x
```

```
fun square (n:int):int = n * n
```

```
(define (square n) (* n n))
```

$$\text{square} \coloneqq \lambda x^{\mathbb{Z}}.(x * x)$$

Evaluation

The rules also tell us how to evaluate or compute terms.
The general rule is:

- *introduction* plus *elimination* computes to *substitution*.

$$\frac{\Gamma, x\colon A \vdash b\colon B \qquad \Gamma \vdash a\colon A}{\Gamma \vdash (\lambda x^{A}.b)(a) \rightarrow_{\beta} b[a/x]}$$

Here $b[a/x]$ means b with a substituted for x.

For historical reasons, this is called β-reduction.

$$\text{square}(2) \equiv (\lambda x^{\mathbb{Z}}.(x * x))(2) \rightarrow_{\beta} (x * x)[2/x] \equiv 2 * 2$$

Functions of many variables

A function of two variables can be represented as a function of one variable which returns a function of another variable.

$$\text{foo} := \lambda x^{\mathbb{Z}}.\Big(\lambda y^{\mathbb{Z}}.(2 * x + y * y)\Big)$$

$$\begin{array}{rcl} \text{foo}(3)(1) & \rightarrow_{\beta} & \big(\lambda y^{\mathbb{Z}}.(2 * x + y * y)\big)[3/x](1) \\ & \equiv & \big(\lambda y^{\mathbb{Z}}.(2 * 3 + y * y)\big)(1) \\ & \rightarrow_{\beta} & (2 * 3 + y * y)[1/y] \\ & \equiv & (2 * 3 + 1 * 1) \end{array}$$

This is called currying (after Haskell Curry).

Functions of many variables

A simplified notation for abstractions:

$$\begin{aligned}\text{foo} &:= \lambda x^{\mathbb{Z}}.\Big(\lambda y^{\mathbb{Z}}.(2 * x + y * y)\Big)\\ &\equiv \lambda x^{\mathbb{Z}} y^{\mathbb{Z}}.(2 * x + y * y)\end{aligned}$$

And for types, → associates to the right:

$$A \to B \to C \quad \text{means} \quad A \to (B \to C)$$

And for application:

$$\text{foo}(3)(1) \rightsquigarrow \text{foo } 3\ 1$$

That is, juxtaposition means application, which associates to the left:

$$\text{foo } 3\ 1 \quad \text{means} \quad (\text{foo } 3)\ 1$$

Another example: disjoint unions

$$\frac{\Gamma \vdash A\colon \text{Type} \qquad \Gamma \vdash B\colon \text{Type}}{\Gamma \vdash A + B\colon \text{Type}}$$

$$\frac{\Gamma \vdash a\colon A}{\Gamma \vdash \text{inl}(a)\colon A + B} \qquad \frac{\Gamma \vdash b\colon B}{\Gamma \vdash \text{inr}(b)\colon A + B}$$

$$\frac{\begin{array}{c}\Gamma \vdash C\colon \text{Type}\\ \Gamma \vdash p\colon A + B \qquad \Gamma, x\colon A \vdash c_A\colon C \qquad \Gamma, y\colon B \vdash c_B\colon C\end{array}}{\Gamma \vdash \text{case}(p, x^A.c_A, y^B.c_B)\colon C}$$

Case switching

$$\frac{\begin{array}{c}\Gamma \vdash C\colon \mathrm{Type} \\ \Gamma \vdash p\colon A+B \qquad \Gamma, x\colon A \vdash c_A\colon C \qquad \Gamma, y\colon B \vdash c_B\colon C\end{array}}{\Gamma \vdash \mathrm{case}(p, x^A.c_A, y^B.c_B)\colon C}$$

```
switch(p) {
  if p is inl(x):
    do cA with x
  if p is inr(y):
    do cB with y
}
```

Don’t worry about the exact syntax of “case”. Everyone does it differently, and we’ll mostly use Coq’s syntax (later).

Evaluating case switches

$$\frac{\begin{array}{c}\Gamma \vdash C\colon \mathrm{Type}\ \Gamma \vdash p\colon A+B \\ \Gamma, x\colon A \vdash c_A\colon C \qquad \Gamma, y\colon B \vdash c_B\colon C \qquad \Gamma \vdash a\colon A\end{array}}{\Gamma \vdash \mathrm{case}(\mathrm{inl}(a), x^A.c_A, y^B.c_B) \rightarrow_\beta c_A[a/x]}$$

$$\frac{\begin{array}{c}\Gamma \vdash C\colon \mathrm{Type} \qquad \Gamma \vdash p\colon A+B \\ \Gamma, x\colon A \vdash c_A\colon C \qquad \Gamma, y\colon B \vdash c_B\colon C \qquad \Gamma \vdash b\colon B\end{array}}{\Gamma \vdash \mathrm{case}(\mathrm{inr}(b), x^A.c_A, y^B.c_B) \rightarrow_\beta c_B[b/y]}$$

The unit type

$$\frac{}{\Gamma \vdash \mathrm{unit} : \mathrm{Type}} \qquad \frac{}{\Gamma \vdash \mathrm{tt} : \mathrm{unit}}$$

$$\frac{\Gamma \vdash p : \mathrm{unit} \qquad \Gamma \vdash C : \mathrm{Type} \qquad \Gamma \vdash c : C}{\Gamma \vdash \mathrm{triv}(p, c) : C}$$

If we know how to produce a C using all the possible inputs that can go into a unit, *then we can produce a C from any* unit.

$$\frac{\Gamma \vdash C : \mathrm{Type} \qquad \Gamma \vdash c : C}{\Gamma \vdash \mathrm{triv}(\mathrm{tt}, c) \to_\beta c}$$

When we evaluate the eliminator on a term of canonical form, we obtain the data that went into the eliminator associated to that form.

Polarity

Negative types	Positive types
$A \to B$	$A + B$
$\prod_{x : A} B(x)$	$A \times B$
	unit
	empty
	$\sum_{x : A} B(x)$

- A negative type is characterized by its eliminations.
  1. We use a term by applying it.
  2. We construct a term by saying what it does when applied.
- A positive type is characterized by its introductions.
  1. We construct a term with a constructor.
  2. We use a term by saying what to do with each constructor.

Polarity

Negative types	Positive types
$A \to B$	$A + B$
$\prod_{x:A} B(x)$	$A \times B$
	unit
	empty
	$\sum_{x:A} B(x)$

NB: This is an oversimplification; some or all of these “positive types” could also be presented negatively. But for us, they will be positive.

Types in Coq

Coq uses a type theory called the predicative Calculus of (co)Inductive Constructions. There are only four ways to construct types in Coq.

1. Dependent product (negative).
   - Includes $A \to B$ as a special case; more later
   - Constructed with `fun x => ...`
   - Applied with juxtaposition `f x`
2. Inductive type families (positive).
   - Built with constructors like `inl`, `inr`, `tt`.
   - Eliminated with `match`.
   - More details later.
3. Universes (sorts) like Type (unpolarized).
4. Coinductive type families (negative).

Exercise #1

Exercise

Define the cartesian product $A \times B$ as a positive type.

$$\frac{\Gamma \vdash A\colon \mathrm{Type} \qquad \Gamma \vdash B\colon \mathrm{Type}}{\Gamma \vdash A \times B\colon \mathrm{Type}}$$

$$\frac{\Gamma \vdash a\colon A \qquad \Gamma \vdash b\colon B}{\Gamma \vdash (a, b)\colon A \times B}$$

$$\frac{\Gamma \vdash C\colon \mathrm{Type} \qquad \Gamma \vdash p\colon A \times B \qquad \Gamma, x\colon A, y\colon B \vdash c\colon C}{\Gamma \vdash \mathrm{unpack}(p, x^A\, y^B.c)\colon C}$$

$$\frac{\Gamma \vdash C\colon \mathrm{Type} \qquad \Gamma \vdash a\colon A \qquad \Gamma \vdash b\colon B \qquad \Gamma, x\colon A, y\colon B \vdash c\colon C}{\Gamma \vdash \mathrm{unpack}((a, b), x^A\, y^B.c) \to_\beta c[a/x, b/y]}$$

Projections

For $p\colon A \times B$:

$$\mathrm{fst}(p) \coloneqq \mathrm{unpack}(p, x^A\, y^B.x)\colon A$$

$$\mathrm{snd}(p) \coloneqq \mathrm{unpack}(p, x^A\, y^B.y)\colon B$$

Exercise #2

Exercise

Define the empty type $\emptyset$ as a positive type.

$$\frac{}{\Gamma \vdash \emptyset \colon \mathrm{Type}}$$

(no introduction rule)

$$\frac{\Gamma \vdash p \colon \emptyset \qquad \Gamma \vdash C \colon \mathrm{Type}}{\Gamma \vdash \mathrm{abort}(p) \colon C}$$

(no computation rule)

Structural rules

We also need a few rules for “how to get going” with typing judgments.

$$\frac{\Gamma \vdash A \colon \mathrm{Type}}{\Gamma, x \colon A \vdash x \colon A} \quad \text{start} \quad (x \notin \Gamma)$$

$$\frac{\Gamma \vdash A \colon \mathrm{Type} \qquad \Gamma \vdash b \colon B}{\Gamma, x \colon A \vdash b \colon B} \quad \text{weakening} \quad (x \notin \Gamma)$$

$$\frac{\Gamma \vdash a \colon A \qquad \Gamma \vdash A \leftrightarrow_\beta B}{\Gamma \vdash a \colon B} \quad \text{conversion}$$

($\leftrightarrow_\beta$ is the equivalence relation generated by $\rightarrow_\beta$)

Now you know something!

Definition

The structural rules plus the type constructor $\to$ (and nothing else) form the simply typed lambda calculus “$\lambda_\to$”.

We can of course add other constructors. Sometimes people write $\lambda_{\times\to}$ for $\lambda_\to$ with cartesian products and unit, etc.

Logic in the style of type theory

We can also read a typing judgment

$$x_1 : P_1, \ldots, x_n : P_n \vdash q : Q$$

as a truth judgment

Under hypotheses P_1, P_2, ..., P_n,
the conclusion Q is provable.

Logical connectives

The basic rules tell us how to construct valid truth judgments.
This includes:

1. How to construct new propositions.
2. How to prove such propositions.
3. How to use such propositions to prove other propositions.

Example (Implication)

1. If P and Q are propositions, then so is $P \Rightarrow Q$.
2. If assuming P, we can prove Q, then we can prove $P \Rightarrow Q$.
3. If we can prove P and $P \Rightarrow Q$, then we can prove Q.

Implication

To emphasize this viewpoint, we write Prop rather than Type.

$$\frac{\Gamma \vdash P \colon \mathrm{Prop} \qquad \Gamma \vdash Q \colon \mathrm{Prop}}{\Gamma \vdash (P \Rightarrow Q) \colon \mathrm{Prop}}$$

$$\frac{\Gamma, x \colon P \vdash q \colon Q}{\Gamma \vdash \lambda x^{P}.q \colon P \Rightarrow Q}$$

$$\frac{\Gamma \vdash f \colon P \Rightarrow Q \qquad \Gamma \vdash p \colon P}{\Gamma \vdash f(p) \colon Q}$$

Conjunction

($P \wedge Q$ means “P and Q”)

$$\frac{\Gamma \vdash P \colon \mathrm{Prop} \qquad \Gamma \vdash Q \colon \mathrm{Prop}}{\Gamma \vdash (P \wedge Q) \colon \mathrm{Prop}}$$

$$\frac{\Gamma \vdash p \colon P \qquad \Gamma \vdash q \colon Q}{\Gamma \vdash (p, q) \colon P \wedge Q}$$

$$\frac{\Gamma \vdash R \colon \mathrm{Prop} \qquad \Gamma \vdash s \colon P \wedge Q \qquad \Gamma, x \colon P, y \colon Q \vdash r \colon R}{\Gamma \vdash \mathrm{unpack}(s, x^P\, y^Q . r) \colon R}$$

Disjunction

($P \vee Q$ means “P or Q”)

$$\frac{\Gamma \vdash P \colon \mathrm{Prop} \qquad \Gamma \vdash Q \colon \mathrm{Prop}}{\Gamma \vdash (P \vee Q) \colon \mathrm{Prop}}$$

$$\frac{\Gamma \vdash p \colon P}{\Gamma \vdash \mathrm{inl}(p) \colon P \vee Q} \qquad\qquad \frac{\Gamma \vdash q \colon Q}{\Gamma \vdash \mathrm{inr}(q) \colon P \vee Q}$$

$$\frac{\begin{array}{c} \Gamma \vdash R \colon \mathrm{Prop} \\ \Gamma \vdash s \colon P \vee Q \qquad \Gamma, x \colon P \vdash r_P \colon R \qquad \Gamma, y \colon Q \vdash r_Q \colon R \end{array}}{\Gamma \vdash \mathrm{case}(s, x^P . r_P, y^Q . r_Q) \colon R}$$

Propositions as types

a.k.a. proofs as terms, or the Curry-Howard correspondence

The same rules of programming apply to proving.

Types	$\longleftrightarrow$	Propositions
$A \times B$	$\longleftrightarrow$	P and Q
$A + B$	$\longleftrightarrow$	P or Q
$A \to B$	$\longleftrightarrow$	P implies Q
unit	$\longleftrightarrow$	$\top$ (true)
$\emptyset$	$\longleftrightarrow$	$\bot$ (false)

The program corresponding to a proof computes the “essence” of that proof.

Proof terms

Lemma

For any P and Q, we have $P \Rightarrow (Q \Rightarrow P)$.

Proof.

Assume P. Now if we assume Q, then P by assumption, so $Q \Rightarrow P$. Thus, $P \Rightarrow (Q \Rightarrow P)$. □

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x\colon P \vdash x\colon P}\ \text{(start)}}{x\colon P, y\colon Q \vdash x\colon P}\ \text{(weakening)}}{x\colon P \vdash \lambda y^{Q}.x\colon (Q \Rightarrow P)}\ \text{(introduction)}}{\vdash \lambda x^{P} y^{Q}.x\colon P \Rightarrow (Q \Rightarrow P)}\ \text{(introduction)}$$

Cut elimination

Suppose we prove a lemma:

$$\dfrac{\dfrac{\vdots}{\vdash p\colon P} \qquad \dfrac{\vdots}{\vdash f\colon P \Rightarrow Q}}{\vdash f(p)\colon Q}\ (\text{elimination})$$

Cut elimination

But the way to prove $P \Rightarrow Q$ is to assume P, then prove Q.

$$\dfrac{\dfrac{\vdots}{\vdash p\colon P} \qquad \dfrac{\dfrac{\vdots}{x\colon P \vdash q\colon Q}}{\vdash \lambda x^P.q\colon P \Rightarrow Q}\ (\text{intro})}{\vdash (\lambda x^P.q)(p)\colon Q}\quad (\text{elimination})$$

And since $(\lambda x^P.q)(p) \rightarrow_\beta q[p/x]$, this proof reduces to

$$\dfrac{\dfrac{\dfrac{\vdots}{p\colon P}}{\vdots}}{q[p/x]\colon Q}$$

Negation

We define the negation of P by

$$\neg P := (P \Rightarrow \bot).$$

Lemma

For any P, we have $P \Rightarrow \neg(\neg P)$.

Proof.

Suppose P. To prove $\neg(\neg P)$, suppose $\neg P$. Then since P and $\neg P$, we have a contradiction; hence $\neg(\neg P)$. □

$$\dfrac{\dfrac{x\colon P,\ f\colon (P \Rightarrow \bot)\ \vdash\ f(x)\colon \bot}{x\colon P\ \vdash\ \lambda f^{(P\Rightarrow\bot)}.f(x)\colon ((P \Rightarrow \bot) \Rightarrow \bot)}\ \text{(intro)}}{\vdash\ \lambda x^{P} f^{(P\Rightarrow\bot)}.f(x)\colon P \Rightarrow ((P \Rightarrow \bot) \Rightarrow \bot)}\ \text{(intro)}$$

Intuitionistic logic

BUT the logic we get this way is not quite classical logic:

> *There is no way to write a program to prove $A \vee (\neg A)$.*

What we have is called intuitionistic or constructive logic.
By itself, it is weaker than classical logic. But. . .

1. Many things are still true, when phrased correctly.
2. A weaker logic means a wider validity (in more categories).
3. It is easy to add $A \vee (\neg A)$ as an axiom.
4. There is also a “double-negation translation”. . .

Exercise #3

Exercise

Write a program that proves $\neg(\neg(A \vee (\neg A)))$.

Dependent types

(Back to programming.)

We consider types A_i, B as also expressions, of type “Type”.

Examples

$$\begin{aligned} &\vdash \mathbb{N} \colon \text{Type} \\ A \colon \text{Type},\ x \colon A &\vdash x \colon A \\ A \colon \text{Type},\ B \colon A \to \text{Type},\ x \colon A &\vdash B(x) \colon \text{Type} \\ n \colon \mathbb{N} &\vdash \{k \colon \mathbb{N} \mid k < n\} \colon \text{Type} \\ f \colon \mathbb{R} \to \mathbb{R} &\vdash \{x \colon \mathbb{R} \mid f(x) = 0\} \colon \text{Type} \end{aligned}$$

A judgment $x \colon A \vdash B \colon \text{Type}$, or a term $B \colon A \to \text{Type}$, is a dependent type over A. (The two are interconvertible by λ-abstraction.)

Whence dependent types?

We can construct dependent types as terms of type Type.

Example

Let bool $:=$ unit $+$ unit, and define

$$\begin{aligned} C &:= \lambda b^{\mathrm{bool}}.\mathrm{case}(b, x^{\mathrm{unit}}.\mathbb{Z}, y^{\mathrm{unit}}.\mathbb{R}_{\geq 0}) \\ &: \mathrm{bool} \to \mathrm{Type} \end{aligned}$$

Then

$$\begin{aligned} C(\mathrm{inl}(\mathrm{tt})) &\to_\beta \mathbb{Z} \\ C(\mathrm{inr}(\mathrm{tt})) &\to_\beta \mathbb{R}_{\geq 0} \end{aligned}$$

Dependent products

Given $B \colon A \to \mathrm{Type}$, a term $b \colon \prod_{x \colon A} B(x)$ can be thought of as

1. An A-tuple $(b_x)_{x \colon A}$ with each $b_x \colon B(x)$, or
2. A function b assigning to each $x \colon A$ an element of $B(x)$.

This is a dependently typed function: its *output type* (not just its output *value*) depends on its *input value*.

Remark

If $B(x)$ is independent of x, then $\prod_{x \colon A} B(x)$ reduces to $A \to B$.

Dependent products

$$\frac{\Gamma \vdash A\colon \mathrm{Type} \qquad \Gamma, x\colon A \vdash B\colon \mathrm{Type}}{\Gamma \vdash \prod_{x\colon A} BA \to B\colon \mathrm{Type}}$$

$$\frac{\Gamma, x\colon A \vdash b\colon B}{\Gamma \vdash \lambda x^{A}.b\colon \prod_{x\colon A} BA \to B}$$

$$\frac{\Gamma \vdash f\colon \prod_{x\colon A} BA \to B \qquad \Gamma \vdash a\colon A}{\Gamma \vdash f(a)\colon B[a/x]}$$

$$\frac{\Gamma, x\colon A \vdash b\colon B \qquad \Gamma \vdash a\colon A}{\Gamma \vdash (\lambda x^{A}.b)(a) \to_{\beta} b[a/x]}$$

Dependent sums

Given $B\colon A \to \mathrm{Type}$, a term $p\colon \sum_{x\colon A} B(x)$ consists of

1. a term $a\colon A$, and
2. a term $b\colon B(a)$.

We think of $\sum_{x\colon A} B(x)$ as the disjoint union of the types $B(x)$ over all $x\colon A$.

Remark

If $B(x)$ is independent of x, then $\sum_{x\colon A} B(x)$ reduces to $A \times B$.

Dependent sums

$$\frac{\Gamma \vdash A\colon \mathrm{Type} \qquad \Gamma, x\colon A \vdash B\colon \mathrm{Type}}{\Gamma \vdash \sum_{x\colon A} BA \times B\colon \mathrm{Type}}$$

$$\frac{\Gamma \vdash a\colon A \qquad \Gamma \vdash b\colon B[a/x]}{\Gamma \vdash (a, b)\colon \sum_{x\colon A} BA \times B}$$

$$\frac{\Gamma \vdash C\colon \mathrm{Type} \qquad \Gamma \vdash p\colon \sum_{x\colon A} BA \times B \qquad \Gamma, x\colon A, y\colon B \vdash c\colon C}{\Gamma \vdash \mathrm{unpack}(p, x^A\, y^B.c)\colon C}$$

$$\frac{\begin{array}{c}\Gamma \vdash C\colon \mathrm{Type} \\ \Gamma \vdash a\colon A \qquad \Gamma \vdash b\colon B[a/x] \qquad \Gamma, x\colon A, y\colon B \vdash c\colon C\end{array}}{\Gamma \vdash \mathrm{unpack}((a, b), x^A\, y^B.c) \to_\beta c[a/x, b/y]}$$

Projections

For $p\colon \sum_{x\colon A} B$:

$$\mathrm{pr}_1(p) \coloneqq \mathrm{unpack}(p, x^A\, y^B.x)\colon A$$
$$\mathrm{pr}_2(p) \coloneqq \mathrm{unpack}(p, x^A\, y^B.y)\colon B[\mathrm{pr}_1(p)/x] \leftarrow \text{oops!}$$

$$\frac{\Gamma \vdash C\colon \mathrm{Type} \qquad \Gamma \vdash p\colon \sum_{x\colon A} B \qquad \Gamma, x\colon A, y\colon B \vdash c\colon C}{\Gamma \vdash \mathrm{unpack}(p, x\, y.c)\colon C}$$

We need to allow C to depend on p.

Dependent sums, revised

$$\frac{\Gamma \vdash A \colon \mathrm{Type} \qquad \Gamma, x \colon A \vdash B \colon \mathrm{Type}}{\Gamma \vdash \sum_{x \colon A} BA \times B \colon \mathrm{Type}}$$

$$\frac{\Gamma \vdash a \colon A \qquad \Gamma \vdash b \colon B[a/x]}{\Gamma \vdash (a, b) \colon \sum_{x \colon A} BA \times B}$$

$$\frac{\begin{array}{c} \Gamma, p \colon \sum_{x \colon A} BA \times B \vdash C \colon \mathrm{Type} \\ \Gamma \vdash p \colon \sum_{x \colon A} BA \times B \qquad \Gamma, x \colon A, y \colon B \vdash c \colon C[(x, y)/p] \end{array}}{\Gamma \vdash \mathrm{unpack}(p, x\, y.c) \colon C}$$

$$\frac{\begin{array}{c} \Gamma, p \colon \sum_{x \colon A} BA \times B \vdash C \colon \mathrm{Type} \\ \Gamma \vdash a \colon A \qquad \Gamma \vdash b \colon B[a/x] \qquad \Gamma, x \colon A, y \colon B \vdash c \colon C[(x, y)/p] \end{array}}{\Gamma \vdash \mathrm{unpack}((a, b), x\, y.c) \to_\beta c[a/x, b/y]}$$

Strong eliminators

With dependent types, we need to revise all the eliminators to allow the output type to depend on the input value.

Example

$$\begin{aligned} C &:= \lambda b^{\mathrm{bool}}.\mathrm{case}(b, x^{\mathrm{unit}}.\mathbb{Z}, y^{\mathrm{unit}}.\mathbb{R}_{\geq 0}) \\ &: \mathrm{bool} \to \mathrm{Type} \end{aligned}$$

We need the strong eliminator in order to define

$$\frac{b \colon \mathrm{bool} \vdash \mathrm{case}\left(b, x^{\mathrm{unit}}.(-3), y^{\mathrm{unit}}.\sqrt{2}\right) \colon C(b)}{\vdash \lambda b^{\mathrm{bool}} \ldots \colon \prod_{b \colon \mathrm{bool}} C(b)}$$

Predicate logic

Dependent types + propositions as types = predicate logic!

Types	$\longleftrightarrow$	Propositions
$\prod_{x\colon A} B(x)$	$\longleftrightarrow$	$(\forall x\colon A)P(x)$
$\sum_{x\colon A} B(x)$	$\longleftrightarrow$	$(\exists x\colon A)P(x)$

Universal quantifiers

$$\frac{\Gamma \vdash A\colon \mathrm{Type} \qquad \Gamma, x\colon A \vdash P\colon \mathrm{Prop}}{\Gamma \vdash (\forall x\colon A)P\colon \mathrm{Prop}}$$

$$\frac{\Gamma, x\colon A \vdash p\colon P}{\Gamma \vdash \lambda x^{A}.p\colon (\forall x\colon A)P}$$

$$\frac{\Gamma \vdash f\colon (\forall x\colon A)P \qquad \Gamma \vdash a\colon A}{\Gamma \vdash f(a)\colon P[a/x]}$$

Existential quantifiers

$$\frac{\Gamma \vdash A\colon \mathrm{Type} \qquad \Gamma, x\colon A \vdash P\colon \mathrm{Prop}}{\Gamma \vdash (\exists x\colon A)P\colon \mathrm{Prop}}$$

$$\frac{\Gamma \vdash a\colon A \qquad \Gamma \vdash p\colon P[a/x]}{\Gamma \vdash (a, p)\colon (\exists x\colon A)P}$$

$$\frac{\Gamma \vdash Q\colon \mathrm{Prop} \qquad \Gamma \vdash s\colon (\exists x\colon A)P \qquad \Gamma, x\colon A, p\colon P \vdash q\colon Q}{\Gamma \vdash \mathrm{unpack}(s, x^A p^P.q)\colon Q}$$

Propositions versus types

We now have to face the question:

How do we distinguish the types from the propositions?

Several possibilities:

1. Keep them separate, but analogous. We have sorts “Type” and “Prop”, with separate constructors $\to$ and $\Rightarrow$, $\times$ and $\wedge$, $\prod$ and $\forall$, etc.
2. Make them identical. Every proposition is a type (whose inhabitants are its proof-terms or “witnesses”) and every type is a proposition (the proposition that it is inhabited).
3. Consider propositions as a subclass of types. Usually, they are the types containing at most one inhabitant (“proof-irrelevance”).

Option 1: Separate, but analogous

The Good:

- Flexible: we can later on interpret Prop to be Type or something else.
- Good for verified programming: can automatically discard the proofs of correctness (those in sort Prop) to obtain a working program.
- Can be internalized in weird places like hyperdoctrines and quasitoposes.

The Bad:

- Some seeming redundancy (can be mostly eliminated).
- Doesn't give precise control over what propositions are.
- Need extra axioms and rules to relate Type and Prop; easy to get wrong.

Option 2: Propositions $\equiv$ Types

The Good:

- Irreducibly constructive: every existence "proof" comes with a witness.
- In particular, the "axiom of choice" becomes a theorem.
- Good for studying proofs (different proofs remain distinguishable).

The Bad:

- Can't express the distinction between constructive and nonconstructive existence.
- Questionably compatible with classical mathematics.
- Doesn't correctly interpret in most categories (including homotopy theory).

Option 3: Propositions $\subsetneq$ Types

The Good:

- Distinguishes constructive and nonconstructive existence.
- Interprets correctly into classical mathematics.
- Internalizes in categories (and homotopy theory).
- Some types are automatically propositions (axiom of unique choice).
- Identifies internally the “irrelevant” types to discard.
- Can be implemented “inside” of options 1 or 2.

The Bad:

- Not maximally flexible (doesn’t do hyperdoctrines or quasitoposes).
- All proofs of a proposition are identified (but to distinguish them, we can use the corresponding type).

Propositions versus Types

- Coq chooses option 1: separate but analogous.
- Agda (another computer proof assistant) chooses option 2: make them identical.
- Homotopy type theory uses option 3: propositions are a subclass of types.

Thus, we can do homotopy type theory in Coq or Agda.

Sorts

“Definition”

A type is a syntactic object t which can appear on the right-hand side of a typing judgment $x\colon t$.

“Definition”

A sort is a syntactic object s which can appear on the right-hand side of a typing judgment $t\colon s$, where t is a type.

NB

- These “definitions” are not really standard.
- Logicians say “sort” for what type theorists call a “type”.

Pure type systems

A pure type system is specified by

1. A collection of sorts.
2. A collection of axioms $s_1 : s_2$, for sorts s_1, s_2.
3. The structural rules (start, weakening, conversion), with Type replaced by any sort.
4. A collection of dependency relations (s_1, s_2, s_3), each of which gives a dependent product:

$$\frac{\Gamma \vdash A\colon s_1 \qquad \Gamma, x\colon A \vdash B\colon s_2}{\Gamma \vdash \prod_{x\colon A} B\colon s_3}$$

Can add positive types, with similar sorting relations.

Simply typed lambda calculus, revisited

Example

The simply typed lambda calculus is a pure type system with

1. Two sorts, Type (usually written $*$) and $\Box$.
2. One axiom, Type $: \Box$.
3. One dependency relation $(\text{Type}, \text{Type}, \text{Type})$:

$$\frac{\Gamma \vdash A\colon \text{Type} \qquad \Gamma, x\colon A \vdash B\colon \text{Type}}{\Gamma \vdash \prod_{x\colon A} BA \to B\colon \text{Type}}$$

- With the only relation being $(\text{Type}, \text{Type}, \text{Type})$, there are no nontrivial dependent types.
- $\Box$ is mainly technical here: we need Type $: \Box$ to apply the start rule to type variables. Type is the only inhabitant of $\Box$, and $\Box$ has (and needs) no type.

Polymorphism

If we add to STλC the relation $(\Box, \text{Type}, \text{Type})$, we obtain second-order polymorphic type theory ("$\lambda 2$").

- Type is still the only inhabitant of $\Box$.
- Now types can involve products over Type, e.g.

$$\prod_{A\colon \text{Type}} (A \to A).$$

  An inhabitant of this type consists of, for *every* type A (including itself), a function $A \to A$.
- Seems contradictory in set theory.
- Makes perfect sense in programming, e.g.

$$\lambda A^{\text{Type}} x^{A}.x \; : \; \prod_{A\colon \text{Type}} (A \to A)$$

  the polymorphic identity function.

Higher kinds

Suppose we add the relation $(\Box, \Box, \Box)$.

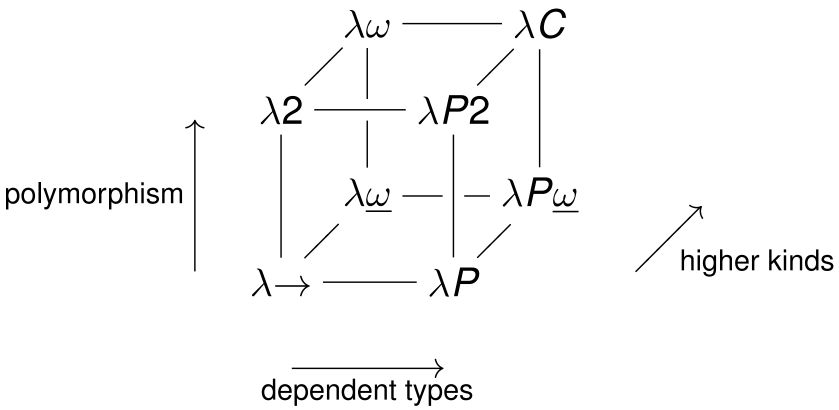
- Now $\Box$ contains other things, like Type $\to$ Type. We call such things kinds, and their inhabitants constructors.
- For example, the operation constructing $A \to B$ out of A and B can now be internalized by a *function*

$$\lambda A^{\mathsf{Type}} B^{\mathsf{Type}}.(A \to B) \;:\; \mathsf{Type} \to (\mathsf{Type} \to \mathsf{Type})$$

With both $(\Box, \mathsf{Type}, \mathsf{Type})$ and $(\Box, \Box, \Box)$, we have higher-order polymorphic type theory ("System $F\omega$" or "$\lambda\omega$").

Dependent types

Finally, adding the relation $(\mathsf{Type}, \Box, \Box)$ gives us dependent types. These eight combinations form the lambda cube:



λC is the impredicative Calculus of Constructions.

Universe levels

If we want to form products and sums over Type, but retain set-theoretic (and homotopy-theoretic) models, we can ramify:

1. Sorts $\mathrm{Type}_0, \mathrm{Type}_1, \mathrm{Type}_2, \ldots$
2. Axioms $\mathrm{Type}_n \colon \mathrm{Type}_{n+1}$
3. Relations $(\mathrm{Type}_n, \mathrm{Type}_m, \mathrm{Type}_k)$ for $k \geq \max(m, n)$.

We may also want a subtyping rule:

$$\frac{\Gamma \vdash A \colon \mathrm{Type}_n}{\Gamma \vdash A \colon \mathrm{Type}_{n+1}}$$

Separate but analogous

Back to types vs. propositions.

Simple predicate logic is the pure type system with

1. Three sorts, Type, Prop, and $\Box$.
2. Two axioms, $\mathrm{Type} \colon \Box$ and $\mathrm{Prop} \colon \Box$.
3. Dependency relations:

$$\begin{array}{lcl}
(\mathrm{Type}, \mathrm{Type}, \mathrm{Type}) & \rightsquigarrow & A \to B \\
(\mathrm{Prop}, \mathrm{Prop}, \mathrm{Prop}) & \rightsquigarrow & P \Rightarrow Q \\
(\mathrm{Type}, \mathrm{Prop}, \mathrm{Prop}) & \rightsquigarrow & (\forall x \colon A), P(x)
\end{array}$$

Separate but analogous

Back to types vs. propositions.

Dependent predicate logic is the pure type system with

1. Three sorts, Type, Prop, and $\Box$.
2. Two axioms, $\text{Type} \colon \Box$ and $\text{Prop} \colon \Box$.
3. Dependency relations:

$$\begin{array}{rcl}
(\text{Type}, \text{Type}, \text{Type}) & \rightsquigarrow & A \to B \\
(\text{Prop}, \text{Prop}, \text{Prop}) & \rightsquigarrow & P \Rightarrow Q \\
(\text{Type}, \text{Prop}, \text{Prop}) & \rightsquigarrow & (\forall x \colon A), P(x) \\
(\text{Type}, \Box, \Box) & \rightsquigarrow & \prod_{x \colon A} P(x)
\end{array}$$

In either case, adding the extra axiom $\text{Prop} \colon \text{Type}$ makes it higher-order.

The Calculus of Constructions

Coq's type theory is the predicative Calculus of Constructions:

1. Sorts Prop and Type_n for $n \geq 1$.
2. Axioms $\text{Prop} \colon \text{Type}_1$ and $\text{Type}_n \colon \text{Type}_{n+1}$.
3. Relations
   - $(\text{Type}_n, \text{Type}_m, \text{Type}_k)$ for $k \geq \max(m, n)$,
   - $(\text{Prop}, \text{Prop}, \text{Prop})$,
   - $(\text{Type}_n, \text{Prop}, \text{Prop})$, and
   - $(\text{Prop}, \text{Type}_n, \text{Type}_n)$.
4. Subtyping

$$\frac{\Gamma \vdash A \colon \text{Type}_n}{\Gamma \vdash A \colon \text{Type}_{n+1}} \qquad \frac{\Gamma \vdash A \colon \text{Prop}}{\Gamma \vdash A \colon \text{Type}_0}$$

Coq notates Type_0 as "Set". Note $\text{Prop} \subseteq \text{Set}$, but $\text{Prop} \notin \text{Set}$.

Universe polymorphism

When doing homotopy type theory in Coq, we generally ignore Prop and Set, and use only the sorts Type$_n$ for $n \geq 1$.

In Coq, all these sorts Type$_n$ are denoted simply “Type”. Coq just checks after each proof that there is a consistent way to assign levels to each occurrence of Type.

Coq is not smart enough to automatically “duplicate” a given definition at more than one universe level. This occasionally causes problems in homotopy type theory. Until Coq is smarter, we can circumvent it by just turning off the consistency checks.

Now you know a lot!

You know basically everything there is to know about Coq’s type theory, except for inductive and coinductive types (next time).