

Type theory and category theory

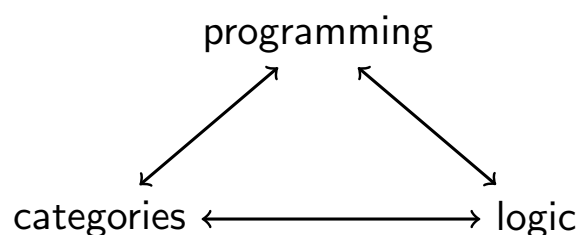
Michael Shulman

<http://www.math.ucsd.edu/~mshulman/hottminicourse2012/>

10 April 2012

The three faces of **homotopy** type theory

- ① A programming language.
 - ② A foundation for mathematics **based on homotopy theory**.
 - ③ A calculus for **$(\infty, 1)$** -category theory.
-
- ① + ② : A computable foundation for **homotopical** mathematics.
 - ② + ③ : A way to internalize **homotopical** mathematics in categories.
 - ① + ③ : A categorical description of programming semantics.



Minicourse plan

- **Today:** Type theory, logic, and category theory
- **Wednesday:** Homotopy theory in type theory
- **Thursday:** Type theory in $(\infty, 1)$ -categories
- **Friday:** Current frontiers

Typing judgments

Type theory consists of rules for manipulating **typing judgments**:

$$(x_1 : A_1), (x_2 : A_2), \dots, (x_n : A_n) \vdash (b : B)$$

- The x_i are variables, while b stands for an arbitrary expression.
- The turnstile \vdash and commas are the “outermost” structure.

This should be read as:

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

The meanings of a typing judgment

$$(x_1 : A_1), (x_2 : A_2), \dots (x_n : A_n) \vdash (b : B)$$

- ① **Programming:** A_i, B are datatypes (`int`, `float`, ...); b is an expression of type B involving variables x_i of types A_i .
- ② **Foundations:** A_i, B are “sets”, b specifies a way to construct an element of B given elements x_i of A_i .
- ③ **Category theory:** A_i, B are objects, b specifies a way to construct a morphism $\prod_i A_i \rightarrow B$.

Type constructors

The rules of type theory come in packages called **type constructors**. Each package consists of:

- ① **Formation:** a way to construct new types.
- ② **Introduction:** ways to construct terms of these types.
- ③ **Elimination:** ways to use them to construct other terms.
- ④ **Computation:** what happens when we follow ② by ③.

Example (Function types)

- ① If A and B are types, then there is a new type B^A .
- ② If $(x : A) \vdash (b : B)$, then $\lambda x. b : B^A$.
- ③ If $a : A$ and $f : B^A$, then $f(a) : B$.
- ④ $(\lambda x. b)(a)$ computes to b with a substituted for x .

Type theory as programming

$\text{square} := \lambda x.(x * x)$

```
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))
```

$\text{square}(2) \equiv (\lambda x.(x * x))(2) \rightsquigarrow 2 * 2$

Type constructors: as foundations

In type theory as a foundation for mathematics:

- All the rules are just “axioms” that give meaning to undefined words like “type” and “term”, out of which we can then build mathematics.
- One usually thinks of “types” as kind of like sets.
- We will consider them as more like “spaces”.

Type constructors: in categories

As a calculus for a cartesian closed category:

- ① If A and B are types, then there is a new type B^A .
 - For objects A and B , there is an exponential object B^A .
- ② If $(x: A) \vdash (b: B)$, then $\lambda x.b: B^A$.
 - Any $Z \times A \rightarrow B$ has an exponential transpose $Z \rightarrow B^A$.
- ③ If $a: A$ and $f: B^A$, then $f(a): B$.
 - The evaluation map $B^A \times A \rightarrow B$.
- ④ $(\lambda x.b)(a)$ computes to b with a substituted for x .
 - The exponential transpose, composed with the evaluation map, yields the original map.

Exactly **the (weak) universal property of an exponential object**.

Inference rules

Type theorists write these rules as follows.

$$\frac{(x: A) \vdash (b: B)}{\vdash (\lambda x.b: B^A)}$$

$$\frac{\vdash (f: B^A) \quad \vdash (a: A)}{\vdash (f(a): B)}$$

The horizontal line means “if the judgments above are valid, so is the one below”. Wide spaces separate multiple hypotheses.

Type theorists also write $A \rightarrow B$ instead of B^A , but this can be confusing when also talking about arrows in a category.

Internal logic

Basic principle

There is a natural correspondence between

- ① **Programming**: ways to build datatypes in a computer
- ② **Foundations**: coherent sets of inference rules for type theory
- ③ **Category theory**: universal properties of objects in a category

Therefore, if we can formalize a piece of mathematics inside of type theory, then

- it can be understood and verified by a computer, and
- it can be internalized in many other categories.

Example: Groups

Informal mathematics

- We have the notion of a **group**: a set G with an element $e \in G$ and a binary operation satisfying certain axioms.
- We can prove theorems about groups, such as that inverses are unique: if $xy = e = xy'$, then $y = y'$.

We can also **formalize** this in ZFC, or in type theory, or in any other precise foundational system.

Example: Group objects

Internal mathematics

- A **group object** in a category is an object G with $e: 1 \rightarrow G$ and $m: G \times G \rightarrow G$, such that some diagrams commute:

$$\begin{array}{ccc} G \times G \times G & \xrightarrow{m \times 1} & G \times G \\ \downarrow 1 \times m & & \downarrow m \\ G \times G & \xrightarrow{m} & G \end{array} \quad \text{etc.}$$

- In sets: a group.
- In topological spaces: a topological group.
- In manifolds: a Lie group.
- In schemes: an algebraic group.
- In rings^{op}: a Hopf algebra.
- In sheaves: a sheaf of groups.

Example: Internalizing groups

Taking the informal notion of a group and formalizing it in type theory, we have a type G and terms

$$\vdash (e : G) \quad (x : G), (y : G) \vdash (x \cdot y : G)$$

satisfying appropriate axioms.

The rules for interpreting type theory in categories give us:

- There is an **automatic** and **general** method which “extracts” or “compiles” the above formalization into the notion of a group object in a category.
- Any theorem about ordinary groups that we can formalize in type theory likewise “compiles” to a theorem about group objects in any category.

We can use “set-theoretic” reasoning with “elements” to prove “arrow-theoretic” facts about arbitrary categories.

Coproduct types

Recall: every type constructor comes with rules for

- ① **Formation:** a way to construct new types.
- ② **Introduction:** ways to construct terms of these types.
- ③ **Elimination:** ways to use them to construct other terms.
- ④ **Computation:** when we follow ② by ③.

Example (Coproduct types)

- ① If A and B are types, then there is a new type $A + B$.
- ② If $a : A$, then $\text{inl}(a) : A + B$. If $b : B$, then $\text{inr}(b) : A + B$.
- ③ If $p : A + B$ and $(x : A) \vdash (c_A : C)$ and $(y : B) \vdash (c_B : C)$, then $\text{case}(p, c_A, c_B) : C$.
- ④ $\text{case}(\text{inl}(a), c_A, c_B)$ computes to c_A with a substituted for x .
 $\text{case}(\text{inr}(b), c_A, c_B)$ computes to c_B with b substituted for y .

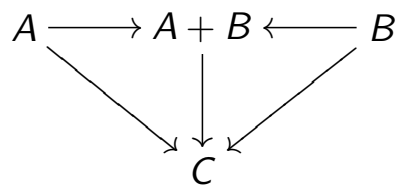
Coproduct types: as programming

- ③ If $p : A + B$ and $(x : A) \vdash (c_A : C)$ and $(y : B) \vdash (c_B : C)$, then $\text{case}(p, c_A, c_B) : C$.

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


Coproduct types: in categories

- 1 If A and B are types, then there is a new type $A + B$.
 - For objects A and B , there is an object $A + B$.
- 2 If $a: A$, then $\text{inl}(a): A + B$. If $b: B$, then $\text{inr}(b): A + B$.
 - Morphisms $\text{inl}: A \rightarrow A + B$ and $\text{inr}: B \rightarrow A + B$.
- 3 If $p: A + B$ and $(x: A) \vdash (c_A: C)$ and $(y: B) \vdash (c_B: C)$, then $\text{case}(p, c_A, c_B): C$.
 - Given morphisms $A \rightarrow C$ and $B \rightarrow C$, we have $A + B \rightarrow C$.
- 4 $\text{case}(\text{inl}(a), c_A, c_B)$ computes to c_A with a substituted for x .
 $\text{case}(\text{inr}(b), c_A, c_B)$ computes to c_B with b substituted for y .
 - The following triangles commute:



Exactly **the (weak) universal property of a coproduct**.

Exercise #1

Exercise

Define the cartesian product $A \times B$.

- 1 If A and B are types, there is a new type $A \times B$.
- 2 If $a: A$ and $b: B$, then $(a, b): A \times B$.
- 3 If $p: A \times B$, then $\text{fst}(p): A$ and $\text{snd}(p): B$.
- 4 $\text{fst}(a, b)$ computes to a , and $\text{snd}(a, b)$ computes to b .

Exercise #2

Exercise

Define the empty type \emptyset .

- ① There is a type \emptyset .
- ②
- ③ If $p: \emptyset$, then $\text{abort}(p): C$ for any type C .
- ④

Aside: Polarity

- A **negative type** is characterized by **eliminations**.
 - We **eliminate** a term in some specified way.
 - We **introduce** a term by saying what it does when eliminated.
 - **Computation** follows the instructions of the introduction.
 - *Examples: function types B^A , products $A \times B$*
- A **positive type** is characterized by **introductions**.
 - We **introduce** a term with specified constructors.
 - We **eliminate** a term by saying how to use each constructor.
 - **Computation** follows the instructions of the elimination.
 - *Examples: coproducts $A + B$, empty set \emptyset*

type theory	\longleftrightarrow	category theory
positive types	\longleftrightarrow	“from the left” universal properties
negative types	\longleftrightarrow	“from the right” universal properties

All universal properties expressible in type theory must be **stable** under products/pullbacks (i.e. adding unused variables).

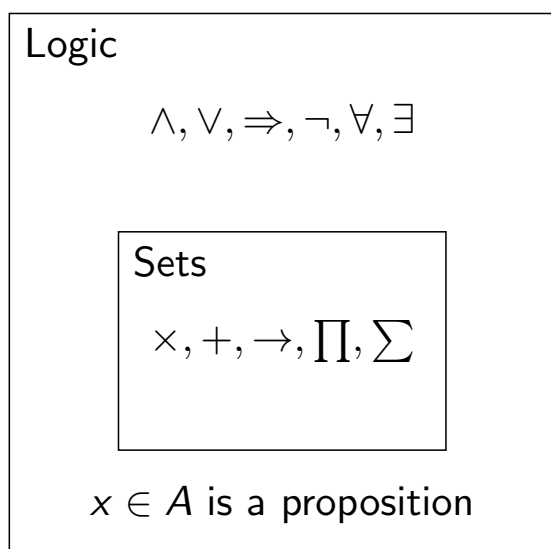
Details that I am not mentioning (yet)

- Uniqueness in universal properties
- η -conversion rules
- Function extensionality
- Dependent eliminators
- Some types have both positive and negative versions
- Universe types (unpolarized)
- Eager and lazy evaluation
- Structural rules
- Coherence issues

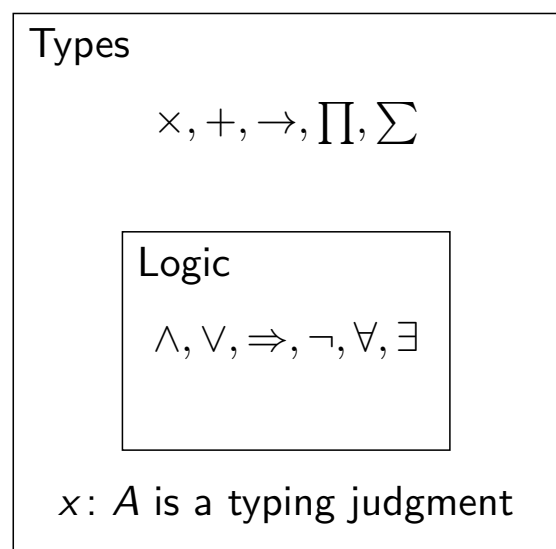
Some of these will come up later.

Type theory versus set theory

Set theory



Type theory



Propositions as some types

Basic principle

We identify a proposition P with the **subsingleton**

$$\{ \star \mid P \text{ is true} \}$$

(That is, $\{\star\}$ if P is true, \emptyset if P is false.)

- To **prove** P is equivalently to exhibit an **element** of it.
- Proofs are just a particular sort of typing judgment:

$$(x_1 : P_1), \dots, (x_n : P_n) \vdash (q : Q)$$

“Under hypotheses P_1, P_2, \dots, P_n ,
the conclusion Q is provable.”

q is a **proof term**, which records how each hypothesis was used.

The Curry-Howard correspondence

Restricted to subsingletons, the rules of type theory tell us **how to construct valid proofs**. This includes:

- ① How to construct new propositions.
- ② How to prove such propositions.
- ③ How to use such propositions to prove other propositions.
- ④ (Computation rules are less meaningful for subsingletons.)

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

Implication

Function types, acting on subsingletons, become **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 .

Conjunction

Cartesian products, acting on subsingletons, become **conjunction**.

- ① If P and Q are propositions, so is “ P and Q ”.
- ② If P is true and Q is true, then so is “ P and Q ”.
- ③ If “ P and Q ” is true, then P is true.
If “ P and Q ” is true, then Q is true.

Proof terms

The proof term

$$(f : P \Rightarrow (Q \text{ and } R)) \vdash (\lambda x. \text{fst}(f(x)) : P \Rightarrow Q)$$

encodes the following informal proof:

Theorem

If P implies Q and R , then P implies Q .

Proof.

- Suppose P .
- Then, by assumption, Q and R .
- Hence Q .
- Therefore, P implies Q . □

This is how **type-checking a program** can **verify a proof**.

Subterminal objects

What does logic look like in a category?

Definition

An object P is **subterminal** if for any object X , there is at most one arrow $X \rightarrow P$.

These are the “truth values” for the “internal logic”.

Examples

- In **Set**: \emptyset (false) and 1 (true).
- In **Set**[→]: false, true, and “in between”.
- In **Set** ^{D} : cosieves in D .
- In **Sh**(X): open subsets of X .

Supports

Problem

Not all operations preserve subsingletons.

- $A \times B$ is a subsingleton if A and B are
- B^A is a subsingleton if A and B are

But:

- $A + B$ is not generally a subsingleton, even if A and B are.

Solution

The **support** of A is a “reflection” of A into subsingletons.

Thus “ P or Q ” means the support of $P + Q$.

I’ll explain the type constructor that does this on Friday.

Intuitionistic logic

We define the **negation** of P by

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

There is no way to prove “ P or $(\neg P)$ ”.

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 It is easy to add “ P or $(\neg P)$ ” as an axiom.
- ③ A weaker logic means a wider validity (in more categories).

Examples

- \mathbf{Set}^D has classical logic $\iff D$ is a groupoid.
- $\mathbf{Sh}(X)$ has classical logic \iff every open set in X is closed.

Exercise #3

Exercise

Write a program that proves $\neg\neg(A \text{ or } \neg A)$.

Details that I am not mentioning

Other ways to interpret logic in type theory:

- Don't require "proposition types" to be subsingletons.
- Keep propositions as a separate "sort" from types.

Predicate logic

For logic we need more than **connectives**

“and”, “or”, “implies”, “not”

we need **quantifiers**:

“for all $x \in X$ ”, “there exists an $x \in X$ such that”

First question

Before forming “there exists an $x \in X$ such that $P(x)$ ”, we need a notion of **predicate**: a “function” P from X to propositions.

Predicates and dependent types

If propositions are subsingleton types, then predicates must be **dependent types**: types that vary over some other type.

A dependent type judgment

$$(x : A) \vdash (B(x) : \text{Type})$$

means that for any particular $x : A$, we have a type $B(x)$. If each $B(x)$ is a subsingleton, then this is a predicate.

Examples of dependent types

$$(y: \text{Year}), (m: \text{Month}) \vdash (\text{Day}(y, m) : \text{Type})$$
$$(x: \mathbb{N}) \vdash (\text{Multiples}(x) : \text{Type})$$
$$(x: \mathbb{N}) \vdash ((x = 0) : \text{Type})$$
$$(x: A), (y: A) \vdash ((x = y) : \text{Type})$$
$$(n: \mathbb{N}), (x: \mathbb{N}), (y: \mathbb{N}), (z: \mathbb{N}) \vdash ((x^n + y^n = z^n) : \text{Type})$$

Universe types

The syntax

$$(x: A) \vdash (B(x) : \text{Type})$$

looks like there is a type called “Type” that $B(x)$ is an element of!

This is called a **universe type**: its *elements* are *types*.

- Can apply λ -abstraction:

$$\lambda x. B(x) : \text{Type}^A$$

- “Type: Type” leads to paradoxes, but we can have a hierarchy

$$\text{Type}_0 : \text{Type}_1 : \text{Type}_2 : \dots$$

Dependent types in categories

In category theory, a dependent type “ $(x: A) \vdash (B(x) : \text{Type})$ ” is:

- 1 A map $B \rightarrow A$, where $B(x)$ is the **fiber** over $x: A$; OR
- 2 A map $A \rightarrow \text{Type}$, where Type is a **universe object**.

The two are related by a pullback:

$$\begin{array}{ccc} B & \longrightarrow & \widetilde{\text{Type}} \\ \downarrow & \lrcorner & \downarrow \\ A & \longrightarrow & \text{Type} \end{array}$$

(Type is the **classifying space** of dependent types).

- B is a **predicate** if $B \rightarrow A$ is **monic**.

Dependent products

A proof of “ $\forall x: A, P(x)$ ” assigns, to each $a: A$, a proof of $P(a)$.
In general, we have the **dependent product**:

- 1 If $(x: A) \vdash (B(x) : \text{Type})$, there is a type $\prod_{x: A} B(x)$.
- 2 If $(x: A) \vdash (b: B(x))$, then $\lambda x. b: \prod_{x: A} B(x)$.
- 3 If $a: A$ and $f: \prod_{x: A} B(x)$, then $f(a): B(a)$.
- 4 $(\lambda x. b)(a)$ computes to b with a substituted for x .

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

- Alternatively: an A -tuple $(f_a)_{a: A}$ with $f_a \in B(a)$.

Remark

If $B(x)$ is independent of x , then $\prod_{x: A} B(x)$ is just B^A .

Dependent sums

A proof of “ $\exists x: A, P(x)$ ” consists of $a: A$, and a proof of $P(a)$.
In general, we have the **dependent sum**:

- 1 If $(x: A) \vdash (B(x) : \text{Type})$, there is a type $\sum_{x: A} B(x)$.
- 2 If $a: A$ and $b: B(a)$, then $(a, b): \sum_{x: A} B(x)$.
- 3 If $p: \sum_{x: A} B(x)$, then $\text{fst}(p): A$ and $\text{snd}(p): B(\text{fst}(p))$.
- 4 $\text{fst}(a, b)$ computes to a and $\text{snd}(a, b)$ computes to b .

$\sum_{x: A} B(x)$ is like the **disjoint union** of $B(x)$ over all $x: A$.

Remark

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

Predicate logic

Types \longleftrightarrow Propositions

$$\begin{array}{l} \prod_{x: A} B(x) \longleftrightarrow \forall x: A, P(x) \\ \sum_{x: A} B(x) \longleftrightarrow \exists x: A, P(x) \end{array}$$

Remarks

- $\prod_{x: A} B(x)$ is a subsingleton if each $B(x)$ is.
- $\sum_{x: A} B(x)$ is not, so we use its support, as with “or”.
- If B is a subsingleton, $\sum_{x: A} B(x)$ is “ $\{x: A \mid B(x)\}$ ”.

Dependent sums and products in categories

- Pullback of a dependent type “ $(y : B) \vdash (P(y) : \text{Type})$ ” along $f : A \rightarrow B$:

$$\begin{array}{ccc} f^*B & \longrightarrow & P \\ \downarrow & \lrcorner & \downarrow \\ A & \xrightarrow{f} & B \end{array}$$

- is **substitution**, yielding “ $(x : A) \vdash (P(f(x)) : \text{Type})$ ”.
- Dependent sum is its **left adjoint** (composition with f).
- Dependent product is its **right adjoint** (in an l.c.c.c).

$$\exists_f \dashv f^* \dashv \forall_f$$

(an insight due originally to Lawvere)

Equality in categories

To formalize mathematics, we need to talk about **equality**.

$$(x : A), (y : A) \vdash ((x = y) : \text{Type})$$

Categorically, this will be a map

$$\begin{array}{c} \text{Eq}_A \\ \downarrow \\ A \times A \end{array}$$

Thinking about fibers leads us to conclude that

Eq_A should be represented by the **diagonal** $A \rightarrow A \times A$.

Equality in type theory

Equality is just another (positive) type constructor.

- 1 For any type A and $a: A$ and $b: A$, there is a type $(a = b)$.
- 2 For any $a: A$, we have $\text{refl}_a: (a = a)$.
- 3

$$\frac{\begin{array}{l} (x: A), (y: A), (p: (x = y)) \vdash (C(x, y, p)) : \text{Type} \\ (x: A) \vdash (d(x)) : C(x, x, \text{refl}_x) \end{array}}{(x: A), (y: A), (p: (x = y)) \vdash (J(d; x, y, p)) : C(x, y, p)}.$$

- 4 $J(d; a, a, \text{refl}_a)$ computes to $d(a)$.

(On Friday: a general framework which produces these rules.)

Homotopical equality

Two Big Important Facts

- 1 The rules **do not imply** that $(x = y)$ is a subsingleton!
- 2 Diagonals $A \rightarrow A \times A$ in higher categories **are not monic!**

Conclusions

- Types naturally form a higher category.
- Type theory naturally has models in higher categories.

