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.

$$\begin{tikzcd} & \text{programming} \arrow[dl, leftrightarrow] \arrow[dr, leftrightarrow] & \\ \text{categories} \arrow[rr, leftrightarrow] & & \text{logic} \end{tikzcd}$$

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:

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

- 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, ..., and x_n of type A_n, the expression b has type B.

The meanings of a typing judgment

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

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 $\prod_i A_i \to B$.

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 $(x\colon A) \vdash (b\colon B)$, then $\lambda x.b\colon B^A$.
   - Any $Z \times A \to B$ has an exponential transpose $Z \to B^A$.
3. If $a\colon A$ and $f\colon B^A$, then $f(a)\colon B$.
   - The evaluation map $B^A \times A \to B$.
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.

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

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

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

Type theorists also write $A \to B$ 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 is a category is an object G with $e\colon 1 \to G$ and $m\colon G \times G \to G$, such that some diagrams commute:

$$\begin{tikzcd} G \times G \times G \arrow[r, "m\times 1"] \arrow[d, "1\times m"'] & G \times G \arrow[d, "m"] \\ G \times G \arrow[r, "m"'] & G \end{tikzcd} \qquad \text{etc.}$$

- 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

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

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 $a \colon A$, then $\mathrm{inl}(a) \colon A + B$. If $b \colon B$, then $\mathrm{inr}(b) \colon A + B$.
3. If $p \colon A + B$ and $(x \colon A) \vdash (c_A \colon C)$ and $(y \colon B) \vdash (c_B \colon C)$, then $\mathrm{case}(p, c_A, c_B) \colon C$.
4. $\mathrm{case}(\mathrm{inl}(a), c_A, c_B)$ computes to c_A with a substituted for x. $\mathrm{case}(\mathrm{inr}(b), c_A, c_B)$ computes to c_B with b substituted for y.

Coproduct types: as programming

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

```
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 $a\colon A$, then $\mathrm{inl}(a)\colon A+B$. If $b\colon B$, then $\mathrm{inr}(b)\colon A+B$.
    - Morphisms $\mathrm{inl}\colon A \to A+B$ and $\mathrm{inr}\colon B \to A+B$.
3. If $p\colon A+B$ and $(x\colon A) \vdash (c_A\colon C)$ and $(y\colon B) \vdash (c_B\colon C)$, then $\mathrm{case}(p, c_A, c_B)\colon C$.
    - Given morphisms $A \to C$ and $B \to C$, we have $A+B \to C$.
4. $\mathrm{case}(\mathrm{inl}(a), c_A, c_B)$ computes to c_A with a substituted for x. $\mathrm{case}(\mathrm{inr}(b), c_A, c_B)$ computes to c_B with b substituted for y.
    - The following triangles commute:

$$\begin{tikzcd}
A \arrow[r] \arrow[dr] & A+B \arrow[d] & B \arrow[l] \arrow[dl] \\
 & C &
\end{tikzcd}$$

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 $a\colon A$ and $b\colon B$, then $(a,b)\colon A \times B$.
3. If $p\colon A \times B$, then $\mathrm{fst}(p)\colon A$ and $\mathrm{snd}(p)\colon B$.
4. $\mathrm{fst}(a,b)$ computes to a, and $\mathrm{snd}(a,b)$ 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
Logic: $\wedge, \vee, \Rightarrow, \neg, \forall, \exists$	Types: $\times, +, \rightarrow, \prod, \sum$
Sets: $\times, +, \rightarrow, \prod, \sum$	Logic: $\wedge, \vee, \Rightarrow, \neg, \forall, \exists$
$x \in A$ is a proposition	$x : A$ is a typing judgment

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:

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

"Under hypotheses $P_1, P_2, \ldots, P_n$,
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)
$\emptyset$	$\longleftrightarrow$	$\bot$ (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

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

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. $\square$

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 $X \to P$.

These are the "truth values" for the "internal logic".

Examples

- In **Set**: $\emptyset$ (false) and 1 (true).
- In $\mathbf{Set}^{\to}$: false, true, and "in between".
- In $\mathbf{Set}^{D}$: cosieves in D.
- In $\mathbf{Sh}(X)$: 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

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

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

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

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

$$(y\colon \text{Year}), (m\colon \text{Month}) \ \vdash \ (\text{Day}(y,m) : \text{Type})$$

$$(x\colon \mathbb{N}) \ \vdash \ (\text{Multiples}(x) : \text{Type})$$

$$(x\colon \mathbb{N}) \ \vdash \ ((x = 0) : \text{Type})$$

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

$$(n\colon \mathbb{N}), (x\colon \mathbb{N}), (y\colon \mathbb{N}), (z\colon \mathbb{N}) \ \vdash \ ((x^n + y^n = z^n) : \text{Type})$$

Universe types

The syntax

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

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:

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

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

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

Dependent types in categories

In category theory, a dependent type "$(x\colon A) \vdash (B(x) : \mathrm{Type})$" is:

1. A map $B \to A$, where $B(x)$ is the fiber over $x\colon A$; OR
2. A map $A \to \mathrm{Type}$, where Type is a universe object.

The two are related by a pullback:

$$\begin{tikzcd} B \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \widetilde{\mathrm{Type}} \arrow[d] \\ A \arrow[r] & \mathrm{Type} \end{tikzcd}$$

(Type is the classifying space of dependent types).

- B is a predicate if $B \to A$ is monic.

Dependent products

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

1. If $(x\colon A) \vdash (B(x) : \mathrm{Type})$, there is a type $\prod_{x\colon A} B(x)$.
2. If $(x\colon A) \vdash (b : B(x))$, then $\lambda x.b\colon \prod_{x\colon A} B(x)$.
3. If $a\colon A$ and $f\colon \prod_{x\colon A} B(x)$, then $f(a)\colon B(a)$.
4. $(\lambda x.b)(a)$ computes to 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 $(f_a)_{a\colon A}$ with $f_a \in B(a)$.

Remark

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

Dependent sums

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

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

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

Remark

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

Predicate logic

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

Remarks

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

Dependent sums and products in categories

- Pullback of a dependent type "$(y : B) \vdash (P(y) : \mathrm{Type})$" along $f : A \to B$:

$$\begin{tikzcd} f^*B \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & P \arrow[d] \\ A \arrow[r, "f"'] & B \end{tikzcd}$$

  is substitution, yielding "$(x : A) \vdash (P(f(x)) : \mathrm{Type})$".
- 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.

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

Categorically, this will be a map

$$\begin{tikzcd} \mathrm{Eq}_A \arrow[d] \\ A \times A \end{tikzcd}$$

Thinking about fibers leads us to conclude that

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

Equality in type theory

Equality is just another (positive) type constructor.

1. For any type A and $a\colon A$ and $b\colon A$, there is a type $(a = b)$.
2. For any $a\colon A$, we have $\mathsf{refl}_a\colon (a = a)$.
3. $$\frac{\begin{array}{lcll}(x\colon A), (y\colon A), (p\colon (x = y)) & \vdash & (C(x,y,p) & : \mathsf{Type}) \\ (x\colon A) & \vdash & (d(x) & : C(x,x,\mathsf{refl}_x))\end{array}}{\begin{array}{lcll}(x\colon A), (y\colon A), (p\colon (x = y)) & \vdash & (J(d;x,y,p) & : C(x,y,p)).\end{array}}$$
4. $J(d;a,a,\mathsf{refl}_a)$ 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 $A \to A \times A$ in higher categories are not monic!

Conclusions

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

$$\begin{tikzcd}
\boxed{\text{Homotopy theory}} \arrow[r, shift left, "\text{homotopy (type theory)}"] & \boxed{\text{Type theory}} \arrow[l, shift left, "\text{(homotopy type) theory}"]
\end{tikzcd}$$