Homotopy Type Theory

the confluence of logic and space

tslil
2018/11/30
1. Foundations

2. Type Theory

3. Homotopy Type Theory

4. Univalence
Foundations
Motivation

• Why should there be a ‘the’ foundation?
  • Euclid – who cares if \( l \ ‘is’ \{(x, y) \in \mathbb{R} \times \mathbb{R} \mid \ldots\}\)
  • Arithmetic – who cares if \(2 \in 3\)
• Set Theory™ does not understand structures and their equalities:
  • We can ask bad questions: \(|7|, \text{succ}(\mathbb{C}), \pi \cap e \in \mathbb{Q}, \ldots\)
  • groups \(G \cong H\leadsto “everything \ that’s \ true \ of \ G \ is \ true \ of \ H”\)
  • vs \(\emptyset \in G\)
• In Set Theoretic foundations like ZF:
  • Technically can’t construct anything
  • Infeasible to actually work in the foundations
  • Nothing about structures to be gained by doing so

Our foundations should reflect how we do mathematics
A technical argument by a trusted author, which is hard to check and looks similar to arguments known to be correct, is hardly ever checked in detail.

~ V. V.

A foundation amenable to

\{ computer verification, proof assistance \}

Proofs are programmes, and may be run!
Type Theory
Instead of ‘everything is set’, why not have different types which reflect the qualitatively different partitions into which we naturally place things?

- $\mathbb{N}$: type of natural numbers
- $\mathbb{Z}$: type of integers
- $\text{Mon}$: type of monoids
- $0$: the ‘empty’ type
- $1$: the ‘singleton’ type
- $U$: the ‘universe’ type
Basics

• Types have terms, $0 : \mathbb{N}, 1 : \mathbb{N}, \ldots$, and terms belong to a unique type.
• Given types $A, B$, we can form more types: $A \times B, A + B, A \rightarrow B, \ldots$
• These new types behave as one might expect, if we have $a : A, b : B$ then $(a, b) : A \times B$.
• Types come equipped with certain functions:
  - $\text{succ} : \mathbb{N} \rightarrow \mathbb{N}$,
  - $\text{pr}_A : A \times B \rightarrow A$,
  - $\text{app}_{A,B} : A \times (A \rightarrow B) \rightarrow B$.

Note: the statement ‘$a : A$’ is not a proposition, it may not be proven or disproven, it is data.
An elaboration

Some types are **inductive**:  

- To define a function from the type $A + B$ it suffices to perform case analysis,  
- If $0$ is the nullary sum then to define a function from $0$ it suffices.  
- $\text{ind}^{A + B}_C : (A \rightarrow C) \times (B \rightarrow C) \rightarrow ((A + B) \rightarrow C)$  
- $\text{ind}^0_C : 0 \rightarrow C$  
- $\text{ind}^N_C : C \times (C \rightarrow C) \rightarrow (\mathbb{N} \rightarrow C)$  
- (Surprise?) To define a function from the type $A \times B$ it suffices to define it on all pairs $(a, b)$. 

7/23
From a humble proposition

What does it mean to work ‘in’ this system, and ‘prove’ things?

**Propositions as types (P.A.T.)**

Encode propositions as types \( P \) and proofs as terms \( p : P \).
Warming up

Let’s implement one of the basic tools of propositional logic,

**A DeMorgan Law**

\[ (\neg P \land (\neg Q)) \implies \neg(P \lor Q) \]

**The translation**

DeMorgan : \[(\neg A \times (\neg B)) \rightarrow \neg(A + B)\]
We want to build a term of type \(((\neg A) \times (\neg B)) \rightarrow \neg (A + B)\)

- It’s enough to define it on input \(\text{inp} : (\neg A) \times (\neg B)\), or \((n_a, n_b)\) where \(n_a \equiv \pi_1(\text{inp}) : \neg A\) and \(n_b \equiv \pi_2(\text{inp}) : \neg B\)
- Now \(\text{DeMorgan}(n_a, n_b) : \neg (A + B) \equiv (A + B) \rightarrow 0\)
- So again it’s enough to define it on input \(z : A + B\), but by induction it’s enough to define it on \(a : A\) and \(b : B\) separately
- Thus we must give \(\text{DeMorgan}(n_a, n_b)(a) : 0\), so the puzzle is to inhabit \(0\) with the context

\[
a : A, n_a : A \rightarrow 0, n_b : B \rightarrow 0
\]

- Just apply \(n_a\) to \(a\)!
We want to build a term of type \(((\neg A) \times (\neg B)) \rightarrow \neg (A + B)\).

Putting everything together

\[
\text{DeMorgan} \equiv (n_a, n_b) \mapsto \text{case}_{A,B}(a \mapsto n_a(a), b \mapsto n_b(b))
\]

where \text{case} is the induction for +.

This is a small taste of what working in type theory is like.
How do we express a predicate over a type?

Take P.A.T. literally,

**A first pass**
A predicate $P$ on $A$ should be something that gives a type $P(a)$ for $a : A$ which is inhabited if $P(a)$ is true.

**Dependent types**
A predicate on $A$ should be a term $P : A \to U$
Is that all? Well it depends...

**Definition**

We allow ourselves to form the type \( A \rightarrow U \). Terms \( B \) of this type are *dependent types* or *type families* varying over \( A \).

We also extend the product forming operation:

**Definition**

Given \( A : U \) and \( B : A \rightarrow U \), we define

- the *dependent product*, \( \prod (a : A), B(a) \), to be the type comprising terms \( f : (a : A) \rightarrow B(a) \)
- the *dependent sum*, \( \sum (a : A), B(a) \), to be the type comprising terms \( (a, b : B(a)) \)
Propositions as types, or, a pronunciation guide

Idea: encode logic statements as types, proofs as terms.

<table>
<thead>
<tr>
<th>Logic</th>
<th>Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proposition on $A$</td>
<td>$A \rightarrow U$</td>
</tr>
<tr>
<td>proof of $P(a)$</td>
<td>$p : P(a)$</td>
</tr>
<tr>
<td>$A \implies B$</td>
<td>$A \rightarrow B$</td>
</tr>
<tr>
<td>$A \land B$</td>
<td>$A \times B$</td>
</tr>
<tr>
<td>$A \lor B$</td>
<td>$A + B$</td>
</tr>
<tr>
<td>$\forall a[P(a)]$</td>
<td>$\prod(a : A), P(a)$</td>
</tr>
<tr>
<td>$\exists a[P(a)]$</td>
<td>$\sum(a : A), P(a)$</td>
</tr>
</tbody>
</table>

“Proof relevance”
The theorem of choice

Classically

\[ \forall a \in A[\exists b \in B_a[R(a, b)]] \implies \exists c \in (\Pi_A B_a)[\forall a \in A[R(a, c(a))]] \]

Typically

\[ \text{ac: } \prod_{(a : A)} \sum_{(b : B)} R(a, b) \rightarrow \sum_{(c : \prod_{(a : A), B})} \prod_{(a : A)} R(a, c(a)) \]

This is true!

In fact, the conclusion is equivalent to the antecedent.
Homotopy Type Theory
How should we express equality?

P.A.T. \( \rightsquigarrow \) for \( a, b : A \) there is a type \( a =_A b \).

Equality is detected by relations \( R : A \to (A \to U) \), but these must be reflexive.

“All equations are lies...or useless”

Reflexivity: \( \text{refl}_a : a =_A a \)

Universal property:

\[
\prod(a : A), R(a, a, \text{refl}_a) \simeq \prod(a, b : A), \prod(p : a =_A b), R(a, b, p)
\]
It’s $\infty$ all the way down

**Definition**

With $a, b, c : A$, $p : a =_A b$, $q : b =_A c$ we can construct

- $p^{-1} : b =_A a$
- $p \cdot q : a =_A c$

And we may give terms witnessing

- $\text{lunit} : \text{refl}_a \cdot p = p$
- $\text{linv} : p^{-1} \cdot p = \text{refl}_a$
- $\text{assoc} : (p \cdot q) \cdot r = p \cdot (q \cdot r)$

**Note:** these are not ‘strict’ equalities, we can only prove them as propositions in our type theory.
The analogy

<table>
<thead>
<tr>
<th>Type Theory</th>
<th>Category Theory</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>types $A, B$</td>
<td>$\infty$-groupoids</td>
<td>spaces</td>
</tr>
<tr>
<td>functions $f : A \to B$</td>
<td>$\infty$-functors</td>
<td>continuous maps</td>
</tr>
<tr>
<td>equality $p : a = b$</td>
<td>equivalence</td>
<td>path</td>
</tr>
<tr>
<td>reflexivity $\text{refl}_a : a = a$</td>
<td>identity</td>
<td>constant path</td>
</tr>
<tr>
<td>symmetry $p^{-1} : b = a$</td>
<td>inverse</td>
<td>path reversal</td>
</tr>
<tr>
<td>transitivity $p \circ q : a = c$</td>
<td>composition</td>
<td>path concat.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logic</th>
<th>Topology</th>
</tr>
</thead>
<tbody>
<tr>
<td>propositions</td>
<td>spaces</td>
</tr>
<tr>
<td>implications</td>
<td>continuous maps</td>
</tr>
<tr>
<td>equalities</td>
<td>paths</td>
</tr>
</tbody>
</table>
Instances of the analogy

**Functions are ∞-functors**

We think of types as ∞-groupoids.

A function $f : A \to B$ acts on morphisms:

$$\text{ap}_f(a, b) : (a =_A b) \to (fa =_B fb)$$

**Dependent types are fibrations**

We think of $\text{pr}_1 : (\sum (a : A), B) \to A$ as a fibration.

A path $p : a =_A a'$ in the base space acts on fibres:

$$\text{transport}_{A,B}(a, a', p) : B(a) \to B(a')$$
Univalence
Given $f, g : A \to B$, what should $f = g$ be?

**Definition**
Given $f, g : \prod (a : A), B$ define $f \sim g \equiv \prod (a : A), (fa =_B ga)$ as the type of homotopies between $f$ and $g$.

“a proof of a predicate is determined by those elements for which the predicate holds”

**Definition**
Function extensionality is the axiom that there is a term

$$\text{funext} : f \sim g \to f = g.$$
Type extensionality

The analogous question for types
Given \( A, B : U \), what should \( A =_U B \) be?

We have a notion of equivalence for types \( A \cong B \), and a canonical term \( \text{idtoeqv} : (A =_U B) \to (A \cong B) \)

Definition
Univalence is the axiom that \( \text{idtoeqv} \) is an equivalence. We name its quasi-inverse

\[ \text{ua} : (A \cong B) \to (A =_U B). \]
Some repercussions

Facts

• Univalence implies function extensionality
• Consistent to assume (Voevodsky’s model in sSet)
• Consequently, for any proposition \( P : U \to U \) and witness \( A \simeq B \), one cannot show \( P(A) \) but not \( P(B) \).

**METATHEORY**

• By the *structure identity principle*, this means that many algebraic things are univalent too.
Univalence for the working mathematician

To get a flavour of the S.I.P., let’s pick our favourite toy algebraic structure, affine schemes ‘sets’ with a binary operation

Magma := \( \sum (A : \text{Set}), A \to (A \to A) \)

Given \((A, \star), (B, \bullet) : \text{Magma}\), what does equality mean?

\[(f : A \simeq B, t : \prod (a, b : A), f(a \star b) =_B (fa) \bullet (fb))\]

\[(f : A \simeq B, s : (f \star =_{B \to (B \to B)} \bullet (f \times f)))\]

\[(q : A =_U B, r : \text{transport}(q, \star) =_{B \to (B \to B)} \bullet)\]

\[p : (A, \star) =_{\text{Magma}} (B, \bullet)\]