Notes on a Review of Type Theory for HoTT

Andrew K. Hirsch

February 1, 2018

1 Philosophy

1. What is type theory?

   (a) A foundation of mathematics in which each object is inseperably associated with a type

   (b) A method of ensuring programs behave correctly

      i. Progress: Well-typed programs don't get stuck

      ii. Preservation: Well-typed programs keep their type

   (c) The two are connected via intuitionism and a computaitonal view of mathematics

2. What is a type?

   (a) An object in a type theory

      i. In "lower-level" type theories, types and terms are distinct classes of objects

      ii. In dependent type theories (such as HoTT), types and terms are the same thing

   (b) Types are used to *describe the behavior* of their associated terms/objects

      i. We say that an object t is an inhabitant of a type τ, and write $t : \tau$

3. Type Theory versus Set Theory

   (a) $t : \tau$ is *not* a proposition

      i. It cannot be predicated upon or negated

   ii. We view every object as inhabitating a type *by its very nature.*

   (b) Judgemental $\equiv$ versus propositional $=$ Equality

      i. Judgemental equality is *not* a proposition, propositional equality *is*
      ii. Judgemental equality "textual," propositional equality is "smarter"

   (c) A Combination of Logic and Ontology

      i. In set theory, set axioms and first-order logic are seperate
      ii. In type theory, types give rise to logic and describe our ontology

         A. Propositions-as-types / Brouwer-Heyting-Kolmogorov (BHK) interpretation / Curry-Howard Isomorphism

2 Types

1. Function Types

   (a) Behavior: "Can be fed a τ, and will then produce a σ."

   (b) Syntax: $\tau \to \sigma$

   (c) Rules:

$$\frac{\Gamma, x : \tau \vdash e : \sigma}{\Gamma \vdash \lambda x : \tau .\ e : \tau \to \sigma} \qquad \frac{\Gamma \vdash f : \tau \to \sigma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash f(e) : \sigma}$$

$$(\lambda x : \tau . e_1)(e_2) \to_\beta e_1[x \mapsto e_2]$$

   Note that I use $e_1[x \mapsto e_2]$ to denote safe substitution of e_2 for x in e_1. I use $\to_\beta$ to denote β *reduction*, which gives meaning to computation in a programming language.

   (d) Logically: "If τ, then σ."

   (e) Categorically: Exponentiation

$$-, B \dashv -^B \qquad \frac{\Gamma, B \vdash C}{\overline{\Gamma \vdash B \to C}}$$

2. Products

(a) Behavior: "Can be constructed from a τ and a σ, can be used in place of either or both."

(b) Syntax: $\tau \times \sigma$

(c) Rules:

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau \times \sigma} \qquad \frac{\Gamma \vdash e_1 : \tau \times \sigma \qquad \Gamma, x : \tau, y : \sigma \vdash e_2 : \chi}{\Gamma \vdash \texttt{match } e_1 \texttt{ with } \langle x, y \rangle \Rightarrow e_2 \texttt{ end} : \chi}$$

$$\begin{array}{lll} \texttt{match } \langle e_1, e_2 \rangle \texttt{ with} & & \\ \langle x, y \rangle \quad \Rightarrow \quad e_3 & & \to_\beta e_3[x \mapsto e_1, y \mapsto e_2] \\ \texttt{end} & & \end{array}$$

Note that I use $e_3[x \mapsto e_1, y \mapsto e_2]$ to denote the *simultaneous* safe substitution of x and y.

(d) Logically "τ and σ"

(e) Categorically: Products

$$\begin{tikzcd}
 & C \arrow[ld, "f"'] \arrow[d, dashed, "{\langle f,g \rangle}"] \arrow[rd, "g"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}$$

3. Sums

(a) Behavior "Behaves as either a τ or a σ, but not both."

(b) Syntax: $\tau + \sigma$

(c) Rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathrm{inl}(e) : \tau + \sigma} \qquad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \mathrm{inr}(e) : \tau + \sigma}$$

$$\frac{\Gamma \vdash e_1 : \tau + \sigma \qquad \Gamma, x : \tau \vdash e_2 : \chi \qquad \Gamma, y : \sigma \vdash e_3 : \chi}{\Gamma \vdash \texttt{match } e_1 \texttt{ with } \mid \mathrm{inl}(x) \Rightarrow e_2 \mid \mathrm{inr}(y) \Rightarrow e_3 \texttt{ end} : \chi}$$

$$\begin{array}{ll} \texttt{match } \mathrm{inl}(e_1) \texttt{ with} & \\ \mathrm{inl}(x) \quad \Rightarrow \quad e_2 & \to_\beta e_2[x \mapsto e_1] \\ \mathrm{inr}(y) \quad \Rightarrow \quad e_3 & \\ \texttt{end} & \end{array}$$

$$\begin{array}{ll} \texttt{match } \mathrm{inr}(e_1) \texttt{ with} & \\ \mathrm{inl}(x) \quad \Rightarrow \quad e_2 & \to_\beta e_3[y \mapsto e_1] \\ \mathrm{inr}(y) \quad \Rightarrow \quad e_3 & \\ \texttt{end} & \end{array}$$

(d) Logically: "τ or σ"

(e) Categorically: Coproducts

$$\begin{tikzcd}
A \arrow[r, "\text{inl}"] & A + B & B \arrow[l, "\text{inr}"'] \\
& C \arrow[ul, "f"] \arrow[u, dashed, "{[f,g]}"] \arrow[ur, "g"'] &
\end{tikzcd}$$

4. Dependent Functions are Infinite Products

   (a) “Infinite” here means “for every member of a type”

   (b) Behavior: “For every x, an $f(x)$ where conceptually $f : \tau \to \text{Type}$”

      i. More generally $\sigma[x]$ i.e. some type σ with x free

   (c) Syntax: $\Pi x : \tau.\ \sigma[x]$, alternatively $(x : \tau) \to \sigma$ We also sometimes write $\prod_{x:\tau} \sigma[x]$ or

$$\prod_{x:\tau} \sigma[x]$$

   if it makes the syntax clearer.

   (d) Rules:

$$\frac{\Gamma, x : \tau \vdash \sigma : U}{\Gamma \vdash \Pi x : \tau.\sigma : U} \qquad \frac{\Gamma, x : \tau \vdash e : \sigma[x]}{\Gamma \vdash \lambda x : \tau.\ e : \Pi x : \tau.\sigma} \qquad \frac{\Gamma \vdash f : \Pi x : \tau.\sigma \qquad \Gamma \vdash e : \tau}{\Gamma \vdash f(e) : \sigma[x \mapsto e]}$$

$$(\lambda x : \tau.e_1)(e_2) \to_\beta e_1[x \mapsto e_2]$$

   Note that I use U to represent a “universe,” or a type of types.

   (e) Note that $\tau \to \sigma$ is just $\Pi x : \tau.\sigma$ where σ does not depend on x.

   (f) Categorically: Deferred for now

5. Dependent Products are Infinite Sums

   (a) Behavior: “For every x, it might be an $\sigma[x]$”

   (b) Syntax: $\Sigma x : \tau.\ \sigma[x]$, alternatively $(x : \tau) \times \sigma$ We also sometimes write $\sum_{x:\tau} \sigma[x]$ or

$$\sum_{x:\tau} \sigma[x]$$

   if it makes the syntax clearer.

(c) Rules:

$$\frac{\Gamma \vdash \tau : U \qquad \Gamma, x : \tau \vdash \sigma : U}{\Gamma \vdash \Sigma x : \tau . \sigma : U}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \sigma[x \mapsto e_1]}{\Gamma \vdash \langle e_1, e_2 \rangle : \Sigma x : \tau . \sigma} \qquad \frac{\Gamma \vdash e_1 : \Sigma x : \tau . \sigma \qquad \Gamma, x : \tau, y : \sigma \vdash e_2 : \chi}{\Gamma \vdash \texttt{match } e_1 \texttt{ with } \langle x, y \rangle \Rightarrow e_2 \texttt{ end} : \chi}$$

$$\begin{array}{l} \texttt{match } \langle e_1, e_2 \rangle \texttt{ with} \\ \langle x, y \rangle \;\; \Rightarrow \;\; e_3 \\ \texttt{end} \end{array} \qquad \to_\beta e_3[x \mapsto e_1, y \mapsto e_2]$$

(d) Note that $\tau \times \sigma$ is just $\Sigma x : \tau . \sigma$ where σ does not depend on x.

(e) Categorically: Deferred for now

6. Inductive Types

   (a) Behavior: "Freely algebra over some generators."

   (b) An example:

```
Inductive Nat : U :=
  | 0 : Nat
  | S : Nat -> Nat.
```

   i. Recursion Principle

$$\prod_{P:\mathrm{Nat}\to U} P(0) \to \left(\prod_{n:\mathrm{Nat}} P(n) \to P(S(n)) \right) \to \prod_{n:\mathrm{Nat}} P(n)$$

   (c) We can "read off" the induction/recursion principal

   (d) To make this precise, we need a type of "well-founded trees" with nodes labelled by a type A and branching factor $B[a]$.

   (e) Logically: A well-founded relation

   (f) Categorically: An F-Algebra for a polynomial functor F

7. (Propositional) Equality Types

   (a) for every type, a type family $\mathrm{Id}_A : A \to A \to U$, so $\mathrm{Id}_A(a, b)$ represents equality between a and b.

   (b) Behavior: Leibniz equality. "If a is true in some statement, so is b. Moreover, $f(a) = f(b)$ for any f."

   (c) Syntax: $a =_A b$. We may drop the subscript A if it is clear from context.

(d) Id_A is the inductive type family generated by constructors $\mathrm{refl}(a) : a =_A a$.

(e) View one: Uniqueness of Identity Proofs (UIP).

  i. If $\pi : a =_A b$, then $\pi = \mathrm{refl}(a)$.

  ii. This is *not* compatable with HoTT!

  iii. True in (unmodified) Agda.

    A. You need to be careful to define dependent pattern matching that does not prove this.

(f) View two: Path induction

  i. Think of $a =_A b$ as a path from a to b in some space.

  ii. Given a family

$$C : \prod_{x:A} (a =_A x) \to U$$

  and

$$c : C(a, \mathrm{refl}(a)),$$

  there is an

$$f : \prod_{x:A} \prod_{\pi:a=x} C(x, \pi)$$

  such that

$$f(a, \mathrm{refl}(a)) \equiv c.$$

  iii. Note: We can prove by path induction on $\pi : x = y$ that $(x, y, p) =_{\sum_{x,y:A} x=y} (x, x, \mathrm{refl}(x))$.

  iv. Homotopically: the space of paths starting at a point x is contractable to the constant loop on x.

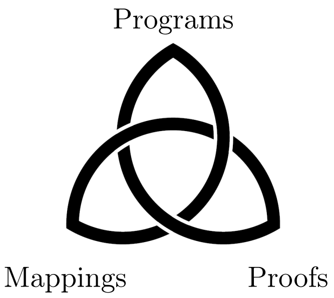
  v. Note: We cannot work over equalities with two fixed endpoints, so we cannot prove that every proof that $x = x$ is by reflexivity.

(g) Categorically: Isomorphism

3 Computational Trinitarianism

> “The Christian doctrine of trinitarianism states that there is one God that is manifest in three persons, the Father, the Son, and the Holy Spirit, who together form the Holy Trinity. The doctrine of computational trinitarianism holds that computation manifests itself in three forms: proofs of propositions, programs of a type, and mappings between structures. These three aspects give rise to three sects of worship: Logic, which gives primacy to proofs and propositions; Languages, which gives primacy to programs and types; Categories, which gives primacy to mappings and structures. The central dogma of computational trinitarianism holds that Logic, Languages, and Categories are but three manifestations of one divine notion of computation. There is no preferred route to enlightenment: each aspect provides insights that comprise the experience of computation in our lives.”
>
> – Robert Harper
> https://existentialtype.wordpress.com/2011/03/27/the-holy-trinity/

1. Programs, proofs, and mappings between structures are all the same thing



2. HoTT rises a question: Should we have Computational Quadrinarianism, with Homotopies being a fourth equal member?

4 If we have time: Hyperdoctrines

1. Categorically, Π- and Σ-types are given meaning via *hyperdoctrines*.

2. Categorical setup: A category per context. Objects are types, and

morphisms $\tau \to \sigma$ in the category for Γ, $\mathcal{C}(\Gamma)$ are e that $\Gamma, x : \tau \vdash e : \sigma$

3. There is an "extension" functor $\pi_\tau^{-1} : \mathcal{C}(\Gamma) \to \mathcal{C}(\Gamma, y : \tau)$

   (a) It ignores the extra τ

4. You might recognize this as a particular fibration. This is the basic setup for categorical logic.

5. Consider the case where $\sum_\tau \dashv \pi_\tau^{-1} \dashv \prod_\tau$

From Adjunction	From Logic and Type Theory
$\begin{array}{c} \pi_\tau^{-1}\sigma \to \rho \\ \hline\hline \sigma \to \displaystyle\prod_\tau \rho \end{array}$	$\begin{array}{c} \Gamma, y : \tau, x : \sigma \vdash \rho \qquad y \text{ is not free in } \Gamma, \sigma \\ \hline\hline \Gamma, \sigma \vdash \displaystyle\prod_{y:\tau} \rho \end{array}$
$\begin{array}{c} \tau\sigma \to \pi_\tau^{-1}\rho \\ \hline\hline \displaystyle\sum_\tau \sigma \to \rho \end{array}$	$\begin{array}{c} \Gamma, y : \tau, x : \sigma \vdash \rho \qquad y \text{ is not free in } \Gamma, \rho \\ \hline\hline \Gamma, \Sigma y : \tau.\ \sigma \vdash \rho \end{array}$

   (a) Note: These are equivalent to the rules for the appropriate constructor/destructor in type theory!