Computational Higher Type Theory (CHTT)

Robert Harper

Lecture Notes of Week 6 by Farzaneh Derakhshan and Di Wang

1 Setting the Scene

Last week, we talked about structural equality and behavioral equality. The development of such reasoning broached the "propositions-as-types" principle (e.g., $z : \mathsf{Eq}_{\mathsf{nat}}(M, N) \gg P(M, N)$ suggests an "equality type" $\mathsf{Eq}_{\mathsf{nat}}(M, N)$), which leads to the exploration of dependent type theory. The development reflects the unification of logic and type theory. Before delving into dependent types, we will first review the history of the "propositions-as-types" principle.

2 Propositions-as-Types

2.1 Two Key Players

We know that computational type theory can be seen as a foundational *theory of truth*, while formal type theory can be viewed as a *theory of proof*. Here we discuss some historical context of these theoretical developments.

Brouwer developed a theory of *truth* based on computation, which was carried forward by Church and Martin-Löf. The theory suggests reasoning about truth semantically by constructing programs. For example, a proposition $A \supset B$ should mean that there is a computable function from evidence for A to evidence for B. The theory was derived from an inchoate understanding of algorithms.

Hilbert developed a theory of formal *proof*, followed by Gentzen, who developed structural proof theory. Rather than semantic, the theory takes a syntactic approach. For example, the judgment $M : A$ is defined inductively and it means that M is a formal *derivation* of A true. This kind of systems are axiomatic and consistency is an important property. However, this theory is meaningless with respect to *truth* by design. Generally speaking, the development of certain formal systems might need to prove theorems similar to the following one to *access* the truth:

> If $M : A$, then $|M| \in A$.

In other words, formal proof is *at most* a sufficient condition for truth. Nevertheless, much effort was devoted to developing formal proof theory.

2.2 Formal Proof in Logic

Historically, there were two flavors of proof systems (i.e., type systems): (i) Hilbert type systems, which are based on combinators, and (ii) Gentzen type systems, which are based on λ-terms. Hilbert's development emphasizes *unconditional truth*. An instance of such systems could be three axioms

$$\alpha \supset (\beta \supset \alpha)$$
$$(\alpha \supset (\beta \supset \gamma)) \supset (\alpha \supset \beta) \supset (\alpha \supset \gamma)$$
$$(\neg\alpha \supset \neg\beta) \supset (\beta \supset \alpha)$$

with *modus ponens* (i.e., from α and $\alpha \supset \beta$, infer β).

Different from Hilbert's approach, Gentzen's development is based on *entailment*, which leads to *hypothetical* reasonings. An example of entailments could be $A\ \mathsf{true} \vdash B\ \mathsf{true}$, saying that suppose A true, one can derive B true. Then the introduction and elimination rules for $\supset$ could be expressed as

$$\frac{A\ \mathsf{true} \vdash B\ \mathsf{true}}{A \supset B\ \mathsf{true}}\ \supset\text{-I} \qquad\qquad \frac{A \supset B\ \mathsf{true} \qquad A\ \mathsf{true}}{B\ \mathsf{true}}\ \supset\text{-E}$$

An interesting fact was discovered during studying proof simplification. If a proof of B true is derived from the proof of $A \supset B$ true and A true, then the proof of $A \supset B$ true has a hypothetical form—suppose A true, then derive B true—therefore one could "plug in" the proof of A true into the places that require A true in the proof of $A \supset B$ true, in order to derive a proof of B true. In other words, proof simplification is based on *substitution*. Coincidentally, λ-calculus, developed by Church, is also based on substitution. Then a formal correspondence, or the *Church-Gentzen correspondence*, is developed between λ-terms and formal proofs. The correspondence is two-fold:

- Pre-existing logics correspond to type systems for λ-terms.
- With Gentzen's input, a notion of proof equivalence is developed.

Nowadays, both formal logics and type systems are given in Gentzen's style, by defining $\Gamma \vdash M : A$ (justified entailment) and $\Gamma \vdash M \equiv N : A$ (justified equivalence).

2.3 Brouwerian Conception

As we already discussed, formal proof theory is only a means of accessing the truth. The Brouwerian principle says instead that truth should be based on computation. Rather than give an inductive definition of $M : A$, one develops the notion of $M \in A$, which means that M is a construction witness of A. More specifically, $M \in A$ means that

1. M evaluates to a value (i.e., a term in a canonical form), and
2. the value obeys the specification given by A.

Similarly, meaning of $M \doteq N \in A$, which stands for equality, is also based on computation.

Based on the theory of truth, a semantic correspondence is developed between propositions and types. This is the so-called "propositions-as-types" principle or "proofs-as-programs" principle. Some examples of the correspondence are shown in Table 1.

Proposition	Type	Proposition	Type
$\top$	$\mathbf{1}$	$A \wedge B$	$A \times B$
$\bot$	$\mathbf{0}$	$A \vee B$	$A + B$
$A \supset B$	$A \to B$	$\neg A = A \supset \bot$	$A \to \mathbf{0}$

Table 1: Propositions-as-types

There is an infamous issue for this theoretical development—there is *not* a proof for the law of excluded middle (i.e., $A \vee \neg A$). However, the loss of excluded middle *does* make sense in the computational setting. We know that $A \vee \neg A$ corresponds to $A + (A \to \mathbf{0})$. If the proposition is true, then the type should have an inhabitant, thus we know the construction witness of either A or $A \to \mathbf{0}$. Suppose A represents the "P versus NP" problem. If we admit excluded middle in the computational setting, then we would get a proof of either P = NP or P $\neq$ NP right away! On the other hand, there are at least two approaches to express the "truth" that A is either true or false:

1. Use a weaker disjunction and in fact, $\neg\neg(A \vee \neg A)$ is provable.
2. Make use of open-endedness. See Howe [1991] for more details.

In the next section, we will explore how the correspondence is extended to quantification $(\forall, \exists)$ and equality.

3 Formal Dependent Type Theory

Our goal is to develop the core **Intentional Type Theory** (**"ITT"**)[1]. Its key idea is the notion of type-indexed families of types (predicates for relations). People also use "dependent types" instead of "family of types" because it *depends* on the family of types. (An example is the family of types $\mathsf{Vec}(M)$ where $M : \mathsf{Nat}$.) In particular, we deal with a type-indexed family which is the identity type $\mathsf{Id}_A(M, N)$ where $M, N \in A$ (The family of types indexed over the type $A \times A$) [2]. We also deal with the generalized types sigma-type (Σ) and pi-type (Π). But first, we need to present the principles that are governing these types.

3.1 Syntactic Judgments:

$$\Gamma \vdash M : A \qquad\qquad \Gamma \vdash M \equiv M' : A$$
$$\Gamma \vdash A\ \mathsf{type} \qquad\qquad \Gamma \vdash A \equiv A'$$

They look like the judgments we had so far, but a little bit more complicated because of the presence of additional judgment forms which are expressing " a type is *relative to the context* "; For example, in this setting, we can write $N : \mathsf{Nat} \vdash \mathsf{Vec}(N)\ \mathsf{type}$ to express that $\mathsf{Vec}(N)$ is a valid type in the context of $N : \mathsf{Nat}$.
Also, we have:

$$\Gamma\ ctxt \qquad\qquad \Gamma \equiv \Gamma'.$$

3.2 Axiomatic Structural Properties

Reflexivity (R) :

$$\frac{}{\Gamma, a : A, \Gamma' \vdash a : A}\ (\mathrm{R})$$

Note that $a : \mathsf{Nat}, b : \mathsf{Seq}(a)$ is a context, where the type of b is dependent on a; the context is ordered now. In the Reflexivity(R) principle, we implicitly say that a is a type in the context of Γ. So, we need a Weakening principle to say that if a is a type in the context Γ, it is also a type in Γ, Γ'.
Weakening (W) :

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash B\ \mathsf{type}}{\Gamma, a : A \vdash B\ \mathsf{type}}\ (\mathrm{W})$$

This principle states that B remains a type, even if we throw extra stuff to Γ. In other words If Γ entails B, then Γ, a also entails B. This holds for every context, so being a little

[1] Warning: details are delicate, and we are going to ignore them
[2] From now on, when we say family, we mean family of types.

hand-wavy we can generalize the principle to:

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash J}{\Gamma, a : A \vdash J}\ (\mathrm{W})$$

Permutation (P) : Order does not matter except when it does!
Contraction (C): Which is an instance of substitution:

$$\frac{\Gamma, a : A, b : B, \Gamma' \vdash J}{\Gamma, c : A, [c, c/a, b]\Gamma' \vdash [c, c/a, b]J}\ (\mathrm{C})$$

Substitution(S):

$$\frac{\Gamma \vdash M : A \qquad \Gamma, a : A, \Gamma' \vdash J}{\Gamma[M/a]\Gamma' \vdash [M/a]J}\ (\mathrm{S})$$

This rule states that judgments are stable under substitution; this corresponds to the transitivity of entailment, and is also called *cut* in Gentzen's system.
These basic structural principles correspond to the idea that variables are treated structurally (vs. substructurally—situations where variables have some restrictions on their use, e.g. Weakening is denied in Relevance logic, first introduced by Nuel Belnap[3].)
Invariance (I):

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash A \equiv A'}{\Gamma \vdash M : A'}\ (\mathrm{I}) \qquad \frac{\Gamma \vdash M \equiv M' : A \qquad \Gamma \vdash A \equiv A'}{\Gamma \vdash M \equiv M' : A'}\ (\mathrm{I})$$

The idea is that equivalent types are equivalent as class of types; if A is definitionally equivalent to A', then the members of A is equivalent to the members of A'. Then the central question is that what is/should be $\Gamma \vdash A \equiv A'$? (Really matters in the *dependent type* setting.) The relation $\Gamma \vdash A \equiv A'$ should at least be reflexive, symmetric, transitive, and compatible with constructions.

Now, that we formalized all the structural rules, we look at Booleans as an example.

3.3 Booleans

As a simple example, we want to axiomatize Booleans; here is our first aproximation:

$$\frac{\Gamma\ ctxt}{\Gamma \vdash \mathsf{Bool}\ \mathsf{type}} \qquad \frac{\Gamma\ ctxt}{\Gamma \vdash \mathsf{Bool} \equiv \mathsf{Bool}}$$

$$\frac{}{\Gamma \vdash \mathsf{tt} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{ff} : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash M : \mathsf{Bool} \qquad \Gamma \vdash P : A \qquad \Gamma \vdash Q : A}{\Gamma \vdash \mathsf{if}(M; P; Q) : A}$$

$$\frac{\Gamma \vdash P : A \qquad \Gamma \vdash Q : A}{\Gamma \vdash \mathsf{if}(\mathsf{ff}; P; Q) \equiv Q : A} \qquad \frac{\Gamma \vdash P : A \qquad \Gamma \vdash Q : A}{\Gamma \vdash \mathsf{if}(\mathsf{tt}; P; Q) \equiv P : A}$$

However, this is not good enough, and there are some issues that we are concerned about:

[3] 1960's- University of Pittsburgh

- **Type Synthesis** (For reasons of decidability): Given the context, we expect to synthesize the type of any term from the term itself. We can do that by adding a label A to the if-clause in the above rules, e.g,

$$\frac{\Gamma \vdash M : \mathsf{Bool} \qquad \Gamma \vdash P : A \qquad \Gamma \vdash Q : A}{\Gamma \vdash \mathsf{if}_A(M; P; Q) : A}$$

- **Dependency** (propositions-as-types): Since we are in the framework of dependent types, the type of something can depend on the type of its terms. So, we generalize the above rules to:

$$\frac{\Gamma, a : \mathsf{Bool} \vdash A\ \mathsf{type} \qquad \Gamma \vdash M : \mathsf{Bool} \qquad \Gamma \vdash P : [\mathsf{tt}/a]A \qquad \Gamma \vdash Q : [\mathsf{ff}/a]A}{\Gamma \vdash \mathsf{if}_{a.A}(M; P; Q) : [M/a]A}$$

  The type of the conditional depends on what we condition on, so we propagate M into a. Similarly, when we are in $\Gamma \vdash P : [\mathsf{tt}/a]A$ we know that we are in the "then" branch, so we propagate tt in A. And when we are in $\Gamma \vdash Q : [\mathsf{ff}/a]A$ we are in the else branch, and we propagate ff in A.
  Another way to read this rule is if A holds for true and A holds for false then A holds for every Boolean (i.e., induction principle for Booleans).

$$\frac{\Gamma \vdash P : [\mathsf{tt}/a]A \qquad \Gamma \vdash Q : [\mathsf{ff}/a]A}{\Gamma \vdash \mathsf{if}_{a.A}(\mathsf{tt}; P; Q) \equiv P : [\mathsf{tt}/a]A} \qquad \frac{\Gamma \vdash P : [\mathsf{tt}/a]A \qquad \Gamma \vdash Q : [\mathsf{ff}/a]A}{\Gamma \vdash \mathsf{if}_{a.A}(\mathsf{ff}; P; Q) \equiv Q : [\mathsf{ff}/a]A}$$

- **Large Elims** (mapping into a collection of all types) Q: What are examples of $a.A$ (Boolean indexed family of types)?
  A: The idea would be to write the type of $\mathsf{if}_{a.\_}(M; 17; \text{"17"})$ as $If(M, Nat, Str)$. But the problem is that in the formal type theory there is a separation between terms and types, so we don't know what sort of thing $\mathsf{If}(M, \mathsf{Nat}, \mathsf{Str})$ is. Thus, we need to extend the syntax with "Large Elim" (LE). In other words, we have to extend what are the types to include conditional branching on term to form a type. In fact, this is not an isolated thing about Booleans, it is about any positive type with the mapping out property.

- **Shannon Expansion** It is expressing the idea of universality of Booleans: In other words Bool is inductively defined as the least type containing true and false, with the mapping out property.

In the next part of this lecture we consider the notions of Π, and Σ (as generalizations of $\to$ and $\times$, respectively).

3.4 Negative Dependent Types, Π and Σ

Here we generalize the negative types to express dependency. Previously, we had:

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash B\ \mathsf{type}}{\Gamma \vdash A \to B\ \mathsf{type}} \qquad \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash B\ \mathsf{type}}{\Gamma \vdash A \times B\ \mathsf{type}}$$

But, in the dependent framework, we want the type of the result to be dependent on the argument you get:

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, a : A \vdash B\ \mathsf{type}}{\Gamma \vdash a : A \to B\ \mathsf{type}} \qquad \frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma, a : A \vdash B\ \mathsf{type}}{\Gamma \vdash a : A \times B\ \mathsf{type}}$$

More conventionally, $a : A \to B$ is known as $\Pi a : A.B$, and $a : A \times B$ is known as $\Sigma a : A.B$.

The Introduction rule is the same to what we had for $\to$, except that we plug in the type of the argument to the type of the result that is dependent on the argument:

$$\frac{\Gamma, a : A \vdash M : B}{\Gamma \vdash \lambda a : A.M : a : A \to B}$$

The following are Elimination and definitional equality rules for $a : A \to B$:

$$\frac{\Gamma \vdash M : a : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M(N) : [N/a]B} \qquad \frac{\Gamma, a : A \vdash M : B \qquad \Gamma \vdash N : A}{\Gamma : (\lambda a : A.M)(N) \equiv [N/a]M[N/a]B}$$

Similarly, we have the following rules for $a : A \times B$:

$$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : [M/a]B \qquad \Gamma, a : A \vdash B\ \mathsf{type}}{\Gamma \vdash \mathsf{pair}_{A,a.B}(M, N) : a : A \times B}$$

$$\frac{\Gamma \vdash M : a : A \times B}{\Gamma \vdash \mathsf{fst}(M) : A} \qquad \frac{\Gamma \vdash M : a : A \times B}{\Gamma : \mathsf{snd}(M)[\mathsf{fst}(M)/a]B}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : [M/a]B}{\Gamma \vdash \mathsf{fst}(\mathsf{pair}_{A,a.B}(M, N)) \equiv M : A} \qquad \frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : [M/a]B}{\Gamma \vdash \mathsf{snd}(\mathsf{pair}_{A,a.B}(M, N)) \equiv N : [M/a]B}$$

3.5 Identity Type

Having the background we gained so far, we are going to answer the question of *How to internalize some idea of "equality"?*

- Definitional Equivalence: This notion is defined for well-typed terms but does not exploit the type. As an example consider $\Gamma \vdash M \equiv \lambda a : A.M(a) : a : A \to B$, and observe that this equality only holds in the type $a : A \to B$. So, it is not sufficient for the purpose of internalizing the equality.
- " Judgmental concepts precede to the type concepts": We can internalize the notion of the map, that we had already, to get identity. We introduce a notion that expresses a binary relation between M an N, and call it $\mathsf{Id}_A(M, N)$:

  $$\frac{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{Id}_A(M, N)\ \mathsf{type}}$$

  But what type is $\mathsf{Id}_A(M, N)$? In some way, it is the least reflexive relation.

  $$\frac{\text{(I)}}{\Gamma \vdash A\ \mathsf{type} \qquad \Gamma \vdash M : A}{\Gamma \vdash \mathsf{refl}_A(M) : \mathsf{Id}_A(M, M)}$$

  So, similar to the definition of the Boolean as the least type containing true and false, we may want to define $\mathsf{Id}_A(M, N)$ as the least reflexive type. However, the difference is that the type Boolean is a single type. In fact, we use the same idea for defining $\mathsf{Id}_A(M, N)$ but instead as a family of types over $A \times A$ (a family whose indexed types

are pairs of elements of A). To express $\mathsf{Id}_A(M,N)$ is actually the least we use the notion of mapping out:

$$\frac{\Gamma \vdash P : \mathsf{Id}_A(M,N) \qquad \Gamma, a:A, b:A, c:\mathsf{Id}_A(a,b) \vdash C\ \mathsf{type} \qquad \Gamma, a:A \vdash Q : [a,a,\mathsf{refl}_A(a)/a,b,c]C}{\Gamma \vdash J_{a,b,c.C}(a.Q)(P) : [M,N,P/a,b,c]C}$$

If we know that M is identical to N in the sense of this identity type by proof term P, then we plug in, all the relevant data, M, N, and P into C. So, that means that we need the premise $\Gamma, a:A, b:A, c:\mathsf{Id}_A(a,b) \vdash C\ \mathsf{type}$, which is called the motive (of induction). The third premise, $\Gamma, a:A \vdash Q : [a,a,\mathsf{refl}_A(a)/a,b,c]C$, says that in order to check the mapping out property it suffices to deal with all the cases where the components of these family are inhabited: the non-empty types that are populated using the Reflexivity principle.
We also have the following rule that expresses what happens when we actually encounter an instance of reflexivity:

$$\frac{\Gamma \vdash M : A \qquad \Gamma, a:A, b:A, c:\mathsf{Id}_A(a,b) \vdash C\ \mathsf{type} \qquad \Gamma, a:A \vdash Q : [a,a,\mathsf{refl}_A(a)/a,b,c]C}{\Gamma \vdash J_{a,b,c.C}(a.Q)(\mathsf{refl}_A(M)) \equiv [M/a]Q : [M,M,\mathsf{refl}_A(M)/a,b,c]C}$$

So far the definition of identity type is established. Now, the question is *If we think of* $\mathsf{Id}_A(M,N)$ *as a relation, what relation it is.*($P : \mathsf{Id}_A(M,N)$ *iff What?)*
It is obviously a reflexive relation, but it also has the following properties:

1. **Symmetry.** $a : \mathsf{Id}_A(M,N) \vdash \mathsf{sym}_{A,M,N}(a) : \mathsf{Id}_A(N,M)$ for all A, M, N.
2. **Transitivity.** $a : \mathsf{Id}_A(M,N), b : \mathsf{Id}_A(N,P) \vdash \mathsf{trans}_{A,M,N,P}(a,b) : \mathsf{Id}_A(M,P)$ for all A, M, N, P.

Exercise 1. *Find* $\mathsf{sym}_{A,M,N}(a)$*, and* $\mathsf{trans}_{A,M,N,P}$*. (Hint: related to Yoneda lemma.)*

Exercise 2. *Show the following equalities:*

- $\mathsf{Id}_{\mathsf{Id}_A(M,M)}(\mathsf{sym}_{A,M,M}(\mathsf{refl}_A(M)), \mathsf{refl}_A(M))$.
- $\mathsf{Id}_{\mathsf{Id}_A(M,N)}(\mathsf{trans}_{A,M,M,N}(\mathsf{refl}_A(M), P), P)$.

References

Douglas J. Howe. On Computational Open-Endedness in Martin-Löf's Type Theory. LICS 91, 1991.