15-819 Homotopy Type Theory
Lecture Notes

Evan Cavallo and Chris Martens

November 4 and 6, 2013

1 Contents

These notes cover Robert Harper's lectures on Homotopy Type Theory from November 4 and 6, 2013. Discussions include the interval type and classical homotopy theory, classification of sertain types as sets, proof irrelevance, and the embedding of classical into constructive logic.

2 The Interval

Definition

Homotopy type theory takes full advantage of the latent ∞-groupoid structure of ITT's identity types by adding new paths. One way we add paths is via the univalence axiom, which introduces new paths between types. We can also directly postulate the existence of types with higher path structure. We will eventually develop a general theory of these *higher inductive types*. For the moment, we consider a simple example, the *interval type*.

$$0_I \xrightarrow{\mathsf{seg}} 1_I$$

The interval I is defined inductively with two traditional constructors, 0_I and 1_I. We think of these as two endpoints of an continuum of points, analogous to the interval $[0, 1]$ of classical analysis. With these points alone, the interval is no different from the type $\mathsf{2}$. In order to complete the definition, we also define a path seg which connects the two endpoints. We thus have the following introduction rules:

$$\frac{}{\Gamma \vdash 0_I : I}\ I\text{-I-0} \qquad \frac{}{\Gamma \vdash 1_I : I}\ I\text{-I-1} \qquad \frac{}{\Gamma \vdash \mathsf{seg} : \mathsf{Id}_I(0_I, 1_I)}\ I\text{-I-seg}$$

In order to find the correct elimination rule for this type, we ask the same question we asked in defining the coproduct: how do we map out of this type? For any type A, what is the form of a map $f : \Pi z{:}I.A$? For the sake of simplicity, let's first consider how to define a map $f : I \to A$. We expect the recursor to have the form

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A \quad ?}{\Gamma, x : I \vdash \mathsf{rec}_I[\_.A](x; M; N; ?) : A}$$

with computation rules

$$\begin{aligned} \mathsf{rec}_I[\_.A](0_I; M; N; ?) &\equiv M \\ \mathsf{rec}_I[\_.A](1_I; M; N; ?) &\equiv N \end{aligned}$$

So far, this is just the recursor for 2. To see what additional information we need, notice that for any map $f : I \to A$ we have $\mathsf{ap}_f(\mathsf{seg}) : \mathsf{Id}_A(f(0_I), f(1_I))$ – the values $f(0_I)$ and $f(1_I)$ have to be related in some way. In other words, we need to specify the way that f acts on the path seg. The full (nondependent) recursor therefore has the form

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A \quad \Gamma \vdash P : \mathsf{Id}_A(M, N)}{\Gamma, x : I \vdash \mathsf{rec}_I[\_.A](x; M; N; P) : A}$$

with the computation rules

$$\begin{aligned} \mathsf{rec}_I[\_.A](0_I; M; N; P) &\equiv M \\ \mathsf{rec}_I[\_.A](1_I; M; N; P) &\equiv N \\ \mathsf{ap}_{\mathsf{rec}_I[\_.A](1_I; M; N; P)}(\mathsf{seg}) &\equiv P \end{aligned}$$

For a dependent function $f : \Pi z{:}I.A$, the values $f(0_I)$ and $f(1_I)$ may have different types. Here, we have $\mathsf{apd}_f(\mathsf{seg}) : 0_I =^{z.A}_{\mathsf{seg}} 1_I$, so the dependent eliminator has the form

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : A \quad \Gamma \vdash P : 0_I =^{z.A}_{\mathsf{seg}} 1_I}{\Gamma, x : I \vdash \mathsf{rec}_I[z.A](x; M; N; P) : A[x/z]}$$

with the computation rules

$$\begin{aligned} \mathsf{rec}_I[z.A](0_I; M; N; P) &\equiv M \\ \mathsf{rec}_I[z.A](1_I; M; N; P) &\equiv N \\ \mathsf{apd}_{\mathsf{rec}_I[z.A](1_I;M;N;P)}(\mathsf{seg}) &\equiv P \end{aligned}$$

One question we should ask ourselves is whether this last computation rule should be definitional. Postulating a definitional equality involving an internally-defined function, apd, is highly unnatural. On the other hand, adding new propositional ruins the computational interpretation of the theory. At this point, we have no satisfactory answer to this question. We will use definitional equality; the HoTT book uses propositional equality. The formal developments in Coq and Agda both use propositional equality, but this is largely an artifact of technical restrictions: it is impossible to add axioms for definitional equalities in these languages.

Describing Paths With the Interval

In classical homotopy theory, paths in a space A are defined as continuous mappings $f : I \to A$ where I is the interval $[0, 1]$. $f(0)$ is the left endpoint of the path, $f(1)$ the right endpoint, and the function gives a way of traveling continuously from one endpoint to the other. In homotopy type theory, paths are a primitive notion, but we can show that the classical definition is equivalent. For any type A, the path space $\Sigma x{:}A.\Sigma y{:}A.\mathsf{Id}_A(x, y)$ is equal to $I \to A$, as shown by the following equivalence:

$$\begin{aligned} &f : (\Sigma x{:}A.\Sigma y{:}A.\mathsf{Id}_A(x, y)) \to (I \to A) \\ &f\ \langle x, \langle y, p\rangle\rangle :\equiv \lambda z.\mathsf{rec}_I[\_.A](z; x; y; p) \end{aligned}$$

$$\begin{aligned} &g : (I \to A) \to (\Sigma x{:}A.\Sigma y{:}A.\mathsf{Id}_A(x, y)) \\ &g\ h :\equiv \langle h(0_I), \langle h(1_I), \mathsf{ap}_h(\mathsf{seg})\rangle\rangle \end{aligned}$$

$$\begin{aligned} &\alpha : \Pi s{:}(\Sigma x{:}A.\Sigma y{:}A.\mathsf{Id}_A(x, y)).\ g(f(s)) = s \\ &\alpha\ s :\equiv \mathsf{refl}_{\Sigma x:A.\Sigma y:A.\mathsf{Id}_A(x,y)}(s) \end{aligned}$$

$$\begin{aligned} &\beta : \Pi h{:}(I \to A).\ f(g(h)) = h \\ &\beta\ h :\equiv \mathsf{funext}(\lambda x{:}I.\mathsf{rec}[z.f(g(h))(x) = h(x)](x; \mathsf{refl}_A(h(0_I)); \mathsf{refl}_A(h(1_I)); \\ &\qquad\qquad \mathsf{refl}_{\mathsf{Id}_A(h(0_I),h(1_I))}(\mathsf{ap}_h(\mathsf{seg})))) \end{aligned}$$

Intuitively, the interval has the shape of a single path, so the image of a function $f : I \to A$ is a path in A.

FUNEXT from the Interval

Interestingly, we can prove function extensionality in ITT if we assume the presence of the interval type. Let $f, g : A \to B$ be two functions and assume $h : \Pi x{:}A.\mathsf{Id}_B(f(x), g(x))$; we want to show $\mathsf{Id}_{A\to B}(f, g)$. To do this, we'll define a function $k : I \to (A \to B)$. In order to do that, we'll first want another function $\tilde{k} : A \to (I \to B)$. This function is defined for every $x : A$ by induction on I:

$$\begin{aligned} \tilde{k}(x)(0_I) &:\equiv f(x) \\ \tilde{k}(x)(1_I) &:\equiv g(x) \\ \mathsf{ap}_{\tilde{k}(x)}(\mathsf{seg}) &:\equiv h(x) \end{aligned}$$

The function k is then defined by $k(t)(x) :\equiv \tilde{k}(x)(t)$. Observe that $k(0_I) \equiv f$ and $k(1_I) \equiv g$. Hence, $\mathsf{ap}_k(\mathsf{seg}) : \mathsf{Id}_{A\to B}(f, g)$.

3 ITT is a theory of sets

Without univalence or higher inductive types, we have no way to constructing paths other than reflexivity. For this reason, we would expect that the types of ITT are *homotopically discrete* – they have no higher path structure. Recall the definition of isSet:

$$\mathsf{isSet}(A) :\equiv \prod_{x,y:A} \prod_{p,q:\mathsf{Id}_A(x,y)} p =_{\mathsf{Id}_A(x,y)} q$$

We will be able to show that most of the basic types in ITT are sets, and that most of the type constructors preserve the property of being a set. Depending on how we define the universe $\mathcal{U}$, it may or may not be possible to prove $\mathcal{U}$ is a set. To prove Π preserves sethood, we will need function extensionality, which is not present in pure ITT. However, it is certianly consistent with pure ITT that all types are sets. In HoTT, on the other hand, we will be able to find types which are provably not sets.

Basic constructs

- $\mathsf{1}$: By a homework exercise, we know that $\mathsf{Id}_{\mathsf{1}}(x, y) \simeq \mathsf{1}$, so for any $x, y : \mathsf{1}$ we have a map $f : \mathsf{Id}_{\mathsf{1}}(x, y) \to \mathsf{1}$ which is an equivalence. Let $x, y : \mathsf{1}$

and $p, q : \mathsf{Id}_{\mathsf{1}}(x, y)$ be given. We know that $f(p) =_{\mathsf{1}} \langle\rangle$ and $f(q) =_{\mathsf{1}} \langle\rangle$, so $f(p) =_{\mathsf{1}} f(q)$. Then $f^{-1}(f(p)) =_{\mathsf{Id}_{\mathsf{1}}(x,y)} f^{-1}(f(q))$ by $\mathsf{ap}_{f^{-1}}$, and the properties of inverses give us that $p =_{\mathsf{Id}_{\mathsf{1}}(x,y)} q$.

- $\mathsf{0}$: Given $x, y : \mathsf{0}$ and $p, q : \mathsf{Id}_{\mathsf{0}}(x, y)$, we can simply abort with x to prove that $p =_{\mathsf{Id}_{\mathsf{0}}(x,y)} q$.
- Π: We assume function extensionality. To prove that $\Pi x{:}A.B_x$ is a set, we only need to know that B_x is a set for every $x{:}A$. Assume this is true, and let $f, g : \Pi x{:}A.B$. We want to show any two $p, q : \mathsf{Id}_{\Pi x:A.B}(f, g)$ are equal. By function extensionality, the type $\mathsf{Id}_{\Pi x:A.B}(f, g)$ is equivalent to $\Pi x{:}A.\mathsf{Id}_{B_x}(f(x), g(x))$, so it suffices to prove any homotopies $h, k : \Pi x{:}A.\mathsf{Id}_{B_x}(f(x), g(x))$ are equal. Applying function extensionality again, it is enough to show $h(x) =_{\mathsf{Id}_{B_x}(f(x),g(x))} k(x)$ for every $x{:}A$. This follows from our assumption that B_x is a set.
- Σ: Analogously with the product type $A \times B$, it is possible to show that $\mathsf{Id}_{\Sigma x:A.B_x}(a, b) \simeq \Sigma p : (\mathsf{fst}\ a = \mathsf{fst}\ b).\ (\mathsf{snd}\ a =_p^{x.B_x} \mathsf{snd}\ b)$. Thus, a path in $\Sigma x{:}A.B_x$ is decomposable into a path in A and a path in B_x for some x; if A and B_x are sets, all such paths will be equal, so we can show that all paths in $\Sigma x{:}A.B_x$ are equal.
- $+$: To prove that $A + B$ is a set, we need to assume A and B are sets. Given $x, y : A + B$, we want to show that any two elements of $\mathsf{Id}_{A+B}(x, y)$ are equal. We can do this by a case analysis on x and y. If $x \equiv \mathsf{inl}(a)$ and $y \equiv \mathsf{inl}(a')$, then $\mathsf{Id}_{A+B}(x, y) \simeq \mathsf{Id}_A(a, a')$, so our theorem follows from the fact that A is a set. The case that $x \equiv \mathsf{inr}(b)$ and $y \equiv \mathsf{inr}(b')$ is symmetric. If $x \equiv \mathsf{inl}(a)$ and $y \equiv \mathsf{inr}(b)$ (or in the reverse case), then the space of paths from x and y is empty, so of course any two paths are equal.
- Nat: We will later discuss Hedberg's theorem, which shows that any type with decidable equality is a set. We leave it as an exercise for the reader to show that Nat has decidable equality.

The universe

We did not go into much detail with demonstrating that the universe is a set, but the gist of it is that we can show it by giving "codes," or abstract syntax trees, for every type in the universe such that they map onto the natural numbers. For example, for base types like Nat we just give a terminal code $\dot{\mathsf{Nat}}$ and for the complex types we can concatenate (the inductive codification of) their codes. Then we give an interpretation function to say the appropriate thing, like $T(\dot{\mathsf{Nat}}) = \mathsf{Nat}$ and $T(a \dot{\to} b) = T(\dot{a}) \to T(\dot{(}b)$.

Identity types

We can show that $\mathsf{Id}_A(x, y)$ is a set if A is a set.

Assumption: A is a set, i.e. there is a term H s.t.

$$H : \Pi x, y{:}A.\Pi p, q{:}\mathsf{Id}_A(x, y).\mathsf{Id}_{\mathsf{Id}_A(x,y)}(p, q)$$

For the sake of making deeply-nested subscripts on identity types more readable, let's introduce a few definitions:

$$\begin{aligned} \mathrm{idA}(x, y) &:= \mathsf{Id}_A(x, y) \\ \mathrm{ididA}(x, y, r, s) &:= \mathsf{Id}_{\mathrm{idA}(x,y)}(r, s) \\ \mathrm{idididA}(x, y, r, s, \alpha, \beta) &:= \mathsf{Id}_{\mathrm{ididA}(x,y,r,s)}(\alpha, \beta) \end{aligned}$$

We need to show that for any x, y, $\mathsf{Id}_A(x, y)$ is a set, i.e. construct a proof term of type

$$\Pi r, s{:}\,\mathrm{idA}(x, y).\Pi\alpha, \beta{:}\,\mathrm{ididA}(x, y, r, s).\,\mathrm{idididA}(\alpha, \beta)$$

Assume:

$$\begin{aligned} u, v &: A \\ r, s &: \mathrm{idA}(u, v) \\ \alpha, \beta &: \mathrm{ididA}(u, v, r, s) \end{aligned}$$

Need to construct a term of type $\mathrm{idididA}(u, v, r, s, \alpha, \beta)$.
First, specialize H to $H'(q) : H(u, v, r, q)$.
We exploit the functoriality of H' to get

$$\begin{aligned} \mathrm{apd}_{H'} &: \Pi q, q'{:}\mathsf{Id}_A(u, v).\Pi\gamma{:}\mathsf{Id}_{-}(q, q').\mathsf{Id}_{-}(\gamma_*(H'(q)), H'(q')) \\ \mathrm{apd}_{H'}(r, s, \alpha) &: \mathsf{Id}_{-}(\alpha_*(H'(r)), H'(s)) \\ \mathrm{apd}_{H'}(r, s, \beta) &: \mathsf{Id}_{-}(\beta_*(H'(r)), H'(s)) \end{aligned}$$

By symmetry and transitivity of identity, we can form a term of type

$$\mathsf{Id}_{-}(\alpha_*(H'(r)), \beta_*(H'(r)))$$

and so we can get transport in the identity

$$\mathsf{Id}_{-}(H'(r) \cdot \alpha, H'(r) \cdot \beta)$$

Because the groupid structure tells us we get a cancellation property (?), this means $\alpha = \beta$.

It is left as an exercise to the reader to construct this term in formal notation.

4 ITT + UA is *not* a set theory

In other words, in homotopy type theory, not all types are sets. In particular, $\mathcal{U}$ is a proper groupoid. There are nontrivial paths between the elements of $\mathcal{U}$.

As an example, we can demonstrate two distinct paths between the booleans 2, one which is based on the identity mapping id taking tt to tt and ff to ff. The other is based on not, taking tt to ff and ff to tt.

not and id are two functions from 2 to 2, and we can show them *equivalent* (exercise). Denote with ua the half of UA that takes us from equivalences to paths. Then $\mathsf{ua}(\mathrm{id})$ and $\mathsf{ua}(\mathrm{not})$ are two *paths* from 2 to 2.

We can now refute that these paths are identifiable, i.e. realize

$$\Pi x{:}2.\mathsf{Id}_2(\mathsf{ua}(\mathrm{id})(x), \mathsf{ua}(\mathrm{not})(x)) \to 0$$

$\mathsf{refl}_2(\mathrm{tt}) : \mathrm{tt} =_2 \mathrm{tt}$ by identity introduction, and by the assumed identity between id and not, we can transport to get a proof that $\mathrm{tt} =_2 \mathrm{ff}$. This can be refuted via the path characterization of sum types seen in a previous lecture, yielding 0.

5 n-types

To foreshadow what's to come: we will eventually consider $\mathrm{isSet}(A)$ a special case of the more general is-n-type(A), specifically

$$\begin{array}{ccc} \mathrm{isSet}(A) & \text{becomes} & \text{is-}0\text{-type}(A) \\ \mathrm{isGpd}(A) & \text{becomes} & \text{is-}1\text{-type}(A) \\ \mathrm{is2Gpd}(A) & \text{becomes} & \text{is-}2\text{-type}(A) \\ \vdots & & \vdots \end{array}$$

The types A for which is-n-type(A) holds will be called the n-types. Roughly, it means that "up a level" we have a set (the identities between identities between ... (n times) become identified).

But before we start climbing the ladder upward, let's go the opposite direction and consider (in some sense) $n = -1, -2$, i.e. what happens if we *take away* structure in the sense of differentiation of identity proofs.

6 Proof Irrelevance

So far, we have taken to heart the idea of *proof relevance* and seen that it can be useful for the *evidence for a proposition* to matter, i.e. to treat the proposition as a type and terms inhabiting that type as useful, meaningful data. For example, the natural numbers form a type Nat, and different "*proofs*" of Nat are different numbers—so of course we care to differentiate them.

Now we will consider the special case of proof *irrelevance*: we can identify certain propositions for which we *do not* distinguishing its proofs, i.e. we can consider any $M, N : A$ for this type A to be equivalent. We will call this property isProp (corresponding to is- -1-type in the table above), and formally we define isProp(A) to be the type

$$\Pi x, y{:}A.\mathsf{Id}_A(x, y)$$

Another word used to describe A with this property is "subsingleton." It is a type with at most one element, up to higher homotopy (i.e. if there are multiple elements then there are paths between them).

A motivation for considering this type arises in the domain of dependently typed programming, wherein we want to consider types (propositions) to be *specificatons* for code. For example, consider specifying a function that takes a (possibly infinite) sequence and returns the first index of the sequence that contains the element 0. A type giving this specification might look like

$$\Pi s{:}\mathsf{Nat} \to \mathsf{Nat}.\ \Sigma i : \mathsf{Nat}.s(i) =_{\mathsf{Nat}} 0$$

...except that if we want the function to be *total*, we need some extra information about the input stream, saying that it actually contains a 0 element:

$$\Pi s{:}(\Sigma t{:}\mathsf{Nat} \to \mathsf{Nat}.\Sigma i{:}\mathsf{Nat}.t(i) =_{\mathsf{Nat}} 0).\ \Sigma i : \mathsf{Nat}.(\pi_1 s)(i) =_{\mathsf{Nat}} 0$$

But it turns out that we are now asking for too much information from our input. The above specification has a constant-time algorithm: the input contains a proof, i.e. a witness that it has a 0 element, which is exactly what we are supposed to return. The function is

$$\lambda s.\langle \pi_1(\pi_2 s), \pi_2(\pi_2 s)\rangle$$

This is not the function we wanted to specify: we had in mind something like an inductive traversal of the sequence, stopping when we find the 0 element and returning a tracked index.

The question for resolving this puzzle is "How do we suppress information in a type?". We would like to still require the input to have a 0 element without making that information available in computation; that is, we are interested in only the *propositional* content of the spec.

One way of suppressing information in a type is Brouwer's idea of using double negation, i.e. to put a $\neg\neg$ in front of s's type.

(Digression: if the stream is infinite, it's not actually clear that we would be able to decide whether it contains a 0; we might be worried that by doubly-negating, we no longer have access to that information. Markov's Principle, from the Russian school of constructivism, states that if a Turing machine can't fail to halt, it must halt; i.e. it takes a form of DNE specialized to Turing machines. Alternatively, we can take the NuPRL route and specify a bound k for the sequence such that we know we will find a 0 if there is one.)

Double negation "kills computational content" in the way that we want, and we can formally state that as the following fact:

isProp($\neg\neg A$) for any A.

$\neg\neg A$ is defined as $(A \to 0) \to 0$

NTS a term inhabiting

$$\Pi x, y{:}(\neg\neg A).\mathsf{Id}_{\neg\neg A}(x, y)$$

In lecture it was stated that this is a simple proof using $\mathsf{abort}_{\_}$. If we have function extensionality available it seems straightforward that in fact *any* negated type $A \to 0$ is a prop:

$$f, g{:}\neg C, x{:}C \vdash \mathsf{abort}_C(f\ x) : \mathsf{Id}_0(f\ x, g\ x)$$

With function extensionality we can turn this into

$$f, g{:}\neg C \vdash \mathrm{funext}(\lambda x.\mathsf{abort}_C(f\ x)) : \mathsf{Id}_{\neg C}(f, g)$$

6.1 Gödel's Double Negation Translation

Brouwer's insight about double negation led to Gödel's discovery of a translation embedding classical logic into constructive logic. The idea is to define $\|-\|$ such that if A is provable classically, $\|A\|$ is provable constructively. We can give this translation as:

$$\begin{aligned} \|1\| &= 1 \\ \|A \wedge B\| &= \|A\| \wedge \|B\| \\ \|0\| &= 0 \\ \|A \vee B\| &= \neg\neg(\|A\| \vee \|B\|) \end{aligned}$$

For implication we have two choices. We can either "just squash" the type, which would be sufficient for information erasure:

$$\|A \supset B\| = \|A\| \supset \|B\|$$

...or we can properly embed classical logic with the translation

$$\|A \supset B\| = \|A\| \supset \neg\neg\|B\|$$

We need the latter definition to recover completeness wrt classical logic, since, remembering that classical logic can be formulated as "constructive logic plus DNE (a double negation elimination rule available in general)," we have

$$\|\neg\neg A \supset A\| = \neg\neg A \supset A$$

with the "just squash" principle, but

$$\|\neg\neg A \supset A\| = \neg\neg A \supset \neg\neg A$$

with the classical embedding, which is provable constructively (it is just an instance of the identity).

With the interpretation of $\neg A$ as a *continuation* accepting a term of type A, this translation coincides with the "continuation-passing transform" for compilers.

7 Hedberg's Theorem

Finally, we will touch briefly on Hedberg's Theorem. Hedberg's Theorem is another way to prove something is a set: it states that a type with *decidable equality* is a set. In other words, If

$$\Pi x, y{:}A.\mathsf{Id}_A(x,y) \vee \neg\mathsf{Id}_A(x,y)$$

then isSet(A).

Proof sketch: decidable equality implies *stable* equality, i.e. $\neg\neg \mathsf{Id}_A(x, y) \supset \mathsf{Id}_a(x, y)$, and stable equality implies sethood.