15-819 Homotopy Type Theory
Lecture Notes

Henry DeYoung and Stephanie Balzer

September 9 and 11, 2013

1 Contents

These notes summarize the lectures on homotopy type theory (HoTT) given by Professor Robert Harper on September 9 and 11, 2013, at CMU. They start by providing a introduction to HoTT, capturing its main ideas and its connection to other related type theories. Then they present intuitionistic propositional logic (IPL), giving both an proof-theoretic formulation as well an order-theoretic formulation.

2 Introduction to homotopy type theory

Homotopy type theory (HoTT) is the subject of a very active research community that gathered at the Institute for Advanced Study (IAS) in 2012 to participate in the Univalent Foundations Program. The results of the program have been recently published in the HoTT Book [1].

2.1 HoTT in a nutshell

HoTT is based on Per Martin-Löf's intuitionistic type theory, which provides a foundation for *intuitionistic mathematics* and which is an extension of *Brouwer's program*. Brouwer viewed mathematical reasoning as a human activity and mathematics as a language for communicating mathematical concepts. As a result, Brouwer perceived the ability of executing a step-by-step procedure or algorithm for performing a *construction* as a fundamental human faculty.

Adopting Brouwer's constructive viewpoint, intuitionistic theories view proofs as the fundamental forms of construction. The notion of *proof relevance* is thus a

characteristic feature of an intuitionistic (or constructive[1]) approach. In the context of HoTT, proof relevance means that proofs become mathematical objects [3]. To fully understand this standpoint, it is necessary to draw a distinction between the notion of a *proof* and the one of a *formal proof* [3, 2]. A formal proof is a proof given in a fixed formal system, such as the axiomatic theory of sets, and arises from the application of the inductively defined rules in that system. Whereas every formal proof is also a proof (assuming soundness of the system) the converse is not true. This fact immediately follows from Gödel's Incompleteness Theorem, which precisely states that there exist true propositions (with a proof), but for which there cannot be given a formal proof, using the rules of the formal system. Unlike conventional formally defined systems, HoTT does not surmise that all possible proofs can be fully circumscribed by its rules, but accepts proofs that cannot be formalized in HoTT. These are exactly the proofs that are considered to be *relevant* and, being treated as mathematical objects, they can be formulated internally as objects of the type theory.

Being based on intuitionistic type theory, HoTT facilitates some form of axiomatic freedom. This means in particular that there exist fewer assumptions that apply globally. For instance, a typical such assumption that is missing in a intuitionistic interpretation is the law of the excluded middle. As put forth by Brouwer, an assumption like the law of the excluded middle does not need to be ruled out entirely, but can be introduced locally, in a proof, if needed. Whether a particular local assumption is needed or not is mainly determined by the actual proof (i.e., construction). A sparing use of global assumptions results in proofs that are based on less assumptions and thus in stronger results overall.

Another distinguishing characteristics of HoTT is that it adopts a *synthetic*, rather than an *analytic* reasoning approach. The differentiation goes back to Lawvere and is best explained by an example. Euclidean geometry, for instance, represents a synthetic approach to geometry as it treats geometric figures, like triangles, lines, and circles, as "things" in themselves rather than sets of points. In an analytical approach based on the Cartesian coordinate system, on the other hand, geometric figures are treated as sets of points in the plane and thus are based on the real numbers. Whereas traditional formulations of homotopy theory are analytic, HoTT is synthetic. The differentiation between a synthetic and an analytic reasoning approach is mainly relevant with regard to the approach's amenability to mechanized reasoning. It turns out that synthetic approaches are easier to mechanize than analytical approaches. This holds true in particular for HoTT: since proofs of equality in HoTT correspond to paths in a space, they are cleaner, shorter, and completely mechanizable.

[1] In this course, intuitionism and constructivism are used interchangeably.

2.2 HoTT in type theory context

HoTT unites homotopy theory with type theory, by embodying Brouwer's intuitionism and drawing from Gentzen's proof theory (see Section 3). It is based on the observation that *types* classify the admissible forms of constructions and thus are programmatically sufficient to encompass all known mathematical constructions. This section briefly sketches how HoTT relates to other existing type theories.

2.2.1 Intensional type theory

Intensional type theory (ITT) is a intuitionistic type theory that serves as the core theory for other type theories. Other type theories are merely extensions of ITT.

2.2.2 Extensional type theory

Extensional type theory (ETT) extends ITT with equality of reflection (ER) and uniqueness of identity proofs (UIP):

$$ETT = ITT + ER + UIP$$

Since types are perceived as sets in ETT, ETT gives rise to a intuitionistic theory of sets.

2.2.3 Homotopy type theory

HoTT extends ITT with higher inductive types (HIT) and the univalence axiom (UA):

$$HoTT = ITT + HIT + UA$$

Since types are perceived as abstract spaces in HoTT, HoTT gives rise to a intuitionistic theory of *weak infinity groupoids*.

3 Intuitionistic propositional logic

What is meant by *intuitionistic* logic? It is a *proof-relevant* logic. One might say its slogan is "logic as if people matter", alluding to Brouwer's principle that mathematics is a social process in which proofs are crucial for communication. Whenever a claim of truth of a proposition is made, it must be accompanied by a proof.

As advanced by Per Martin-Löf, a modern presentation of intuitionistic propositional logic (IPL) distinguishes the notions of *judgment* and *proposition*. A judgment is something that may be known, whereas a proposition is something that sensibly may be the subject of a judgment. For instance, the statement "Every natural number larger than 1 is either prime or can be uniquely factored into a product of primes." is a proposition because it sensibly may be subject to judgment. That the statement is in fact true is a judgment. Only with a proof, however, is it evident that the judgment indeed holds.

Thus, in IPL, the two most basic judgments are $A\ \mathsf{prop}$ and $A\ \mathsf{true}$:

$A\ \mathsf{prop}$	A is a well-formed proposition
$A\ \mathsf{true}$	Proposition A is intuitionistically true, i.e., has a proof.

The inference rules for the prop judgment are called formation rules. The inference rules for the true judgment are divided into classes: *introduction rules* and *elimination rules*.

Following Martin-Löf, the meaning of a proposition A is given by the introduction rules for the judgment $A\ \mathsf{true}$. The elimination rules are dual and describe what may be deduced from a proof of $A\ \mathsf{true}$.

The principle of *internal coherence*, also known as Gentzen's principle of inversion, is that the introduction and elimination rules for a proposition A fit together properly. The elimination rules should be strong enough to deduce all information that was used to introduce A (*local completeness*), but not so strong as to deduce information that might not have been used to introduce A (*local soundness*). In a later lecture, we will discuss internal coherence more precisely, but we can already give an informal treatment.

3.1 Negative fragment of IPL

3.1.1 Conjunction

One familiar group of propositions are the conjunctions. If A and B are well-formed propositions, then so is their conjunction, which we write as $A \wedge B$. This is the content of the formation rule for conjunction: it serves as evidence of the judgment $A \wedge B\ \mathsf{prop}$, provided that there is evidence of the judgments $A\ \mathsf{prop}$ and $B\ \mathsf{prop}$.

$$\frac{A\ \mathsf{prop} \quad B\ \mathsf{prop}}{A \wedge B\ \mathsf{prop}}\ \wedge F$$

We have yet to give meaning to conjunction, however; to do so, we must say how to introduce the judgment that $A \wedge B$ is *true*. As the following rule shows, a verification of $A \wedge B$ consists of a proof of $A\ \mathsf{true}$ paired with a proof of $B\ \mathsf{true}$.

$$\frac{A\ \mathsf{true} \quad B\ \mathsf{true}}{A \wedge B\ \mathsf{true}}\ \wedge I$$

What may we deduce from the knowledge that $A \wedge B$ is true? Because every proof of $A \wedge B\ \mathsf{true}$ ultimately introduces that judgment from a pair of proofs of $A\ \mathsf{true}$ and $B\ \mathsf{true}$, we are justified in deducing $A\ \mathsf{true}$ and $B\ \mathsf{true}$ from any proof of $A \wedge B\ \mathsf{true}$. This leads to the elimination rules for conjunction.

$$\frac{A \wedge B\ \mathsf{true}}{A\ \mathsf{true}}\ \wedge E_1 \qquad\qquad \frac{A \wedge B\ \mathsf{true}}{B\ \mathsf{true}}\ \wedge E_2$$

Internal coherence. As previously mentioned, the principle of internal coherence says that the introduction and elimination rules fit together properly: the elimination rules are strong enough, but not too strong.

If we mistakenly omitted the $\wedge E_2$ elimination rule, then there would be no way to extract the proof of $B\ \mathsf{true}$ that was used in introducing $A \wedge B\ \mathsf{true}$—the elimination rules would be too weak.

On the other hand, if we mistakenly wrote the $\wedge I$ introduction rule as

$$\frac{A\ \mathsf{true}}{A \wedge B\ \mathsf{true}},$$

then there would be no proof of $B\ \mathsf{true}$ present to justify deducing $B\ \mathsf{true}$ with the $\wedge E_2$ rule—the elimination rules would be too strong.

3.1.2 Truth

Another familiar and simple proposition is *truth*, which we write as $\top$. Its formation rule serves as immediate evidence of the judgment $\top\ \mathsf{prop}$, that $\top$ is indeed a well-formed proposition.

$$\frac{}{\top\ \mathsf{prop}}\ \top F$$

Once again, to give meaning to truth we must say how to introduce the judgment that $\top$ is true. $\top$ is a trivially true proposition, and so its introduction rule makes the judgment $\top\ \mathsf{true}$ immediately evident.

$$\frac{}{\top\ \mathsf{true}}\ \top I$$

We should also consider an elimination rule: from a proof of $\top\,\mathsf{true}$, what can we deduce? Since $\top$ is trivially true, any such elimination rule would not increase our knowledge—we put in no information when we introduced $\top\,\mathsf{true}$, so, by the principle of conservation of proof, we should get no information out. For this reason, there is no elimination rule for $\top$, and we can see that its absence is coherent with the introduction rule.

The nullary conjunction. An observation about $\top$ that often proves useful is that $\top$ behaves as a nullary conjunction—a conjunction over the empty set of conjuncts, rather than over a set of two conjuncts.

This observation is reflected in the inference rules. Just as the introduction rule for binary conjunction has two premises (one for each of the two conjuncts), the introduction rule for truth has no premises (one for each of the no conjuncts):

$$\frac{A\,\mathsf{true} \quad B\,\mathsf{true}}{A \wedge B\,\mathsf{true}}\,\wedge I \qquad\qquad \frac{}{\top\,\mathsf{true}}\,\top I$$

Likewise, just as there are two elimination rules for binary conjunction, there are no elimination rules for truth:

$$\frac{A \wedge B\,\mathsf{true}}{A\,\mathsf{true}}\,\wedge E_1 \qquad \frac{A \wedge B\,\mathsf{true}}{B\,\mathsf{true}}\,\wedge E_2 \qquad \text{(no } \top E \text{ rule)}$$

3.1.3 Entailment

The last form of proposition in the negative fragment of IPL is implication. However, to define implication, a different form of judgment is required: entailment (also known as logical consequence or a hypothetical judgment). Entailment is written as

$$\underbrace{A_1\,\mathsf{true}, \ldots, A_n\,\mathsf{true}}_{n \geq 0} \vdash A\,\mathsf{true}\,,$$

and expresses the idea that the judgment $A\,\mathsf{true}$ follows from $A_1\,\mathsf{true}, \ldots, A_n\,\mathsf{true}$. You can think of $A_1\,\mathsf{true}, \ldots, A_n\,\mathsf{true}$ as being assumptions from which the conclusion $A\,\mathsf{true}$ may be deduced. The metavariable Γ is typically used to stand for such a context of assumptions.

We should note that, thus far, the inference rules have been presented in a *local form* in which the context of assumptions was left implicit. It would also be possible to make this context explicit. For example, the introduction rule for conjunction in the two different forms is:

$$\frac{A\,\mathsf{true} \quad B\,\mathsf{true}}{A \wedge B\,\mathsf{true}}\,\wedge I \qquad\qquad \frac{\Gamma \vdash A\,\mathsf{true} \quad \Gamma \vdash B\,\mathsf{true}}{\Gamma \vdash A \wedge B\,\mathsf{true}}\,\wedge I$$

In the remainder of these notes, we will write inference rules in local form whenever possible.

As an entailment, the judgment form satisfies several *structural properties*: reflexivity, transitivity, weakening, contraction, and permutation.

Reflexivity. If the entailment judgment is to express logical consequence, that is, that a conclusion follows from some assumptions, then you must accept a principle of reflexivity,

$$\frac{}{A\ \mathsf{true} \vdash A\ \mathsf{true}}\ \mathsf{R},$$

that an assumption is enough to conclude the same judgment. If you tried to deny this principle, the meaning of an assumption would be unclear.

Transitivity. Dual to reflexivity is a transitivity principle that states that a proof of a conclusion satisfies an assumption of the same judgment.

$$\frac{A\ \mathsf{true} \quad A\ \mathsf{true} \vdash C\ \mathsf{true}}{C\ \mathsf{true}}\ \mathsf{T}$$

Transitivity is a lemma rule. If you prove a lemma ($A\ \mathsf{true}$), then you are justified in using it to prove a theorem that explicitly depends ($A\ \mathsf{true} \vdash C\ \mathsf{true}$); taken together, they are *viewed* as a direct proof of the theorem ($C\ \mathsf{true}$).

Transitivity can also be thought of as proof inlining. Rather than pairing the lemma with the theorem that depends upon it, we could inline the lemma's proof at every point at which the theorem refers to the lemma. The result is a truly direct proof of the theorem.

Weakening. Reflexivity and transitivity are undeniable properties of entailment because they give meaning to assumptions—assumptions are strong enough to prove conclusions (reflexivity), but are only as strong as the proofs they stand for (transitivity). But there are also structural properties that can be denied: weakening, contraction, and permutation. Logics that deny any of these properties are called *substructural logics*.

The principle of weakening says that we can add assumptions to a proof without invalidating that proof:

$$\frac{A\ \mathsf{true}}{B\ \mathsf{true} \vdash A\ \mathsf{true}}\ \mathsf{W}$$

Of course, the new proof is of a weaker statement, but it is nevertheless a valid proof.

Denying weakening leads, in part, to relevance logic. It is called relevance logic because the proofs may not contain the unnecessary, irrelevant assumptions that weakening allows. In this course, we will always have the principle of weakening, however.

Contraction. The principle of contraction says that we are unconcerned about the number of copies of an assumption $A\ \mathsf{true}$; one copy is as good as two:

$$\frac{A\ \mathsf{true}, A\ \mathsf{true} \vdash C\ \mathsf{true}}{A\ \mathsf{true} \vdash C\ \mathsf{true}}\ \mathsf{C}$$

Denying contraction (along with weakening) leads to linear logic, in which we wish to reason about the number of copies of an assumption. This is a powerful way to express consumable resources. In this course, we will always have the principle of contraction, however.

Permutation. The principle of permutation, or exchange, says that the order of assumptions does not matter; we can apply any permutation π to the assumptions and still have a valid proof:

$$\frac{\Gamma \vdash C\ \mathsf{true}}{\pi(\Gamma) \vdash C\ \mathsf{true}}\ \mathsf{P}$$

(Note that it is difficult to state the permutation principle in local rule form.)

Denying permutation (along with weakening and contraction) leads to ordered, or noncommutative, logic. It is a powerful way to express ordered structures, like lists or even formal grammars. In this course, we will always have the principle of permutation, however.

3.1.4 Implication

With the entailment judgment in hand, we can give rules for implication.

Like conjunction, if A and B are well-formed propositions, then so is their implication, which we write as $A \supset B$.

$$\frac{A\ \mathsf{prop} \quad B\ \mathsf{prop}}{A \supset B\ \mathsf{prop}}\ \supset F$$

Once again, to give meaning to implication, we must say how to introduce the judgment $A \supset B\ \mathsf{true}$. To prove $A \supset B\ \mathsf{true}$, we assume $A\ \mathsf{true}$ and prove $B\ \mathsf{true}$.

$$\frac{A\ \mathsf{true} \vdash B\ \mathsf{true}}{A \supset B\ \mathsf{true}}\ \supset I$$

In this way, implication internalizes the entailment judgment as a proposition, while we nonetheless maintain the distinction between propositions and judgments. (As an aside for those familiar with category theory, the relationship between entailment and implication is analogous to the relationship between a mapping and a collection of mappings internalized as an object in some categories.)

This introduction rule for implication is the key distinction between Gentzen-style natural deduction calculus and a Hilbert-style axiomatic calculus. In a Hilbert presentation of IPL, there is no separate notion of entailment, making it difficult to reason hypothetically. Instead, one must contort proofs to make use of several seemingly unmotivated axioms about implication.

Thankfully, we will work with natural deduction and be able to reason hypothetically using the introduction rule. But what does the elimination rule for $A \supset B$ look like? Because every proof of $A \supset B\,\mathsf{true}$ ultimately introduces that judgment from a proof of the entailment $A\,\mathsf{true} \vdash B\,\mathsf{true}$, we might like to write the elimination rule as

$$\frac{A \supset B\,\mathsf{true}}{A\,\mathsf{true} \vdash B\,\mathsf{true}}.$$

This is a valid principle of reasoning, but it turns out to be useful to instead adopt an uncurried form as the actual elimination rule:

$$\frac{A \supset B\,\mathsf{true} \quad A\,\mathsf{true}}{B\,\mathsf{true}}\ \supset E.$$

This rule is sometimes referred to as *modus ponens*.

Internal coherence. These introduction and elimination rules are coherent. The elimination rule is strong enough to recover the entailment that any proof of $A \supset B\mathsf{true}$ ultimately uses in introduction, as the following derivation shows.

$$\frac{\dfrac{A \supset B\,\mathsf{true}}{A\,\mathsf{true} \vdash A \supset B\,\mathsf{true}}\ \mathsf{W} \qquad \dfrac{}{A\,\mathsf{true} \vdash A\,\mathsf{true}}\ \mathsf{R}}{A\,\mathsf{true} \vdash B\,\mathsf{true}}\ \supset E$$

On the other hand, the elimination rule is not too strong because it is just an uncurrying of the inverted introduction rule.

3.2 Positive fragment of IPL

3.2.1 Disjunction

As for conjunction and implication, the disjunction, $A \vee B$, of A and B is a well-formed proposition if both A and B are themselves well-formed propositions.

$$\frac{A\ \mathsf{prop} \quad B\ \mathsf{prop}}{A \vee B\ \mathsf{prop}}\ \vee F$$

A disjunction $A \vee B$ may be introduced in either of two ways: $A \vee B$ is true if A is true or if B is true.

$$\frac{A\ \mathsf{true}}{A \vee B\ \mathsf{true}}\ \vee I_1 \qquad\qquad \frac{B\ \mathsf{true}}{A \vee B\ \mathsf{true}}\ \vee I_2$$

To devise the elimination rule, consider what may we deduce from the knowledge that $A \vee B$ is true. For $A \vee B$ to be true, it must have been ultimately introduced using one of the two introduction rules. Therefore, either A or B is true (or possibly both). The elimination rule allows us to reason by these two cases: If $C\ \mathsf{true}$ follows from $A\ \mathsf{true}$ and also follows from $B\ \mathsf{true}$, then C is true in either case.

$$\frac{A \vee B\ \mathsf{true} \quad A\ \mathsf{true} \vdash C\ \mathsf{true} \quad B\ \mathsf{true} \vdash C\ \mathsf{true}}{C\ \mathsf{true}}\ \vee E$$

3.2.2 Falsehood

The unit of disjunction is falsehood, the proposition that is trivially never true, which we write as $\bot$. Its formation rule is immediate evidence that $\bot$ is a well-formed proposition.

$$\frac{}{\bot\ \mathsf{prop}}\ \bot F$$

Because $\bot$ should never be true, it has no introduction rule. The elimination rule captures *ex falso quodlibet*: from a proof of $\bot\ \mathsf{true}$, we may deduce that *any* proposition C is true because there is ultimately no way to introduce $\bot\ \mathsf{true}$.

$$\frac{\bot\ \mathsf{true}}{C\ \mathsf{true}}\ \bot E$$

Once again, the rules cohere. The elimination rule is very strong, but remains justified due to the absence of any introduction rule for falsehood.

The nullary disjunction. We previously noted that $\top$ behaves as a nullary conjunction. In the same way, $\bot$ behaves as a nullary disjunction. For a binary disjunction, there are two introduction rules, $\vee I_1$ and $\vee I_2$, one for each of the two disjuncts; for falsehood, there are no introduction rules:

$$\frac{A\ \mathsf{true}}{A \vee B\ \mathsf{true}}\ \vee I_1 \qquad \frac{B\ \mathsf{true}}{A \vee B\ \mathsf{true}}\ \vee I_2 \qquad \text{(no } \bot I \text{ rule)}$$

Likewise, for a binary disjunction, there is one elimination rule with a premise for the disjunction and one premise for each of the disjuncts; for falsehood, there is one elimination rule with just a premise for falsehood:

$$\frac{A \vee B\ \mathsf{true} \quad A\ \mathsf{true} \vdash C\ \mathsf{true} \quad B\ \mathsf{true} \vdash C\ \mathsf{true}}{C\ \mathsf{true}}\ \vee E \qquad \frac{\bot\ \mathsf{true}}{C\ \mathsf{true}}\ \bot E$$

4 Order-theoretic formulation of IPL

It is also possible to give an order-theoretic formulation of IPL because entailment is a preorder (reflexive and transitive). We want $A \leq B$ to hold exactly when $A\ \mathsf{true} \vdash B\ \mathsf{true}$. We can therefore devise the order-theoretic formulation with these soundness and completeness goals in mind.

4.1 Conjunction as meet

The elimination rules for conjunction (along with reflexivity of entailment) ensure that $A \wedge B\ \mathsf{true} \vdash A\ \mathsf{true}$ and $A \wedge B\ \mathsf{true} \vdash B\ \mathsf{true}$. To ensure completeness of the order-theoretic formulation, we include the rules

$$\frac{}{A \wedge B \leq A} \qquad \frac{}{A \wedge B \leq B},$$

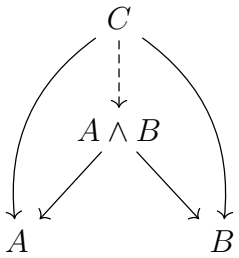
which say that $A \wedge B$ is a lower bound of A and B.

The introduction rule for conjunction ensures that $C\ \mathsf{true} \vdash A \wedge B\ \mathsf{true}$ if both $C\ \mathsf{true} \vdash A\ \mathsf{true}$ and $C\ \mathsf{true} \vdash B\ \mathsf{true}$. Order-theoretically, this is expressed as the rule

$$\frac{C \leq A \quad C \leq B}{C \leq A \wedge B},$$

which says that $A \wedge B$ is as large as any lower bound of A and B. Taken together these rules show that $A \wedge B$ is the greatest lower bound, or meet, of A and B.

Graphically, these order-theoretic rules can be represented with a commuting *product diagram*, where arrows point from smaller to larger elements:



4.2 Truth as greatest element

The introduction rule for $\top$ ensures that $C\,\mathsf{true} \vdash \top\,\mathsf{true}$. Order-theoretically, we have

$$\frac{}{C \leq \top}\,,$$

which says that $\top$ is the greatest, or final, element.

In the proof-theoretic formulation of IPL, we saw that truth $\top$ is the nullary conjunction. We should expect this analogy to hold in the order-theoretic formulation of IPL as well, and it does—the greatest element is indeed the greatest lower bound of the empty set.

4.3 Disjunction as join

The introduction rules for disjunction (along with reflexivity of entailment) ensure that $A\,\mathsf{true} \vdash A \vee B\,\mathsf{true}$ and $B\,\mathsf{true} \vdash A \vee B\,\mathsf{true}$. To ensure completeness of the order-theoretic formulation, we include the rules

$$\frac{}{A \leq A \vee B} \qquad\qquad \frac{}{B \leq A \vee B}\,,$$

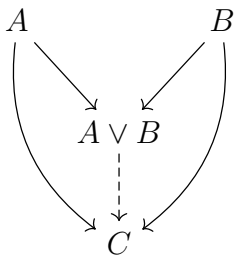
which say that $A \vee B$ is an upper bound of A and B.

The elimination rule for disjunction (along with reflexivity of entailment) ensures that $A \vee B\,\mathsf{true} \vdash C\,\mathsf{true}$ if both $A\,\mathsf{true} \vdash C\,\mathsf{true}$ and $B\,\mathsf{true} \vdash C\,\mathsf{true}$. Order-theoretically, we have the corresponding rule

$$\frac{A \leq C \quad B \leq C}{A \vee B \leq C}\,,$$

which says that $A \vee B$ is as small as any upper bound of A and B. Taken together these rules show that $A \vee B$ is the least upper bound, or join, of A and B.

Graphically, this is captured by a commuting *coproduct diagram*:



4.4 Falsehood as least element

The elimination rule for falsehood (along with reflexivity of entailment) ensures that $\bot\ \mathsf{true} \vdash C\ \mathsf{true}$. The order-theoretic counterpart is the rule

$$\frac{}{\bot \leq C}\,,$$

which says that $\bot$ is the least, or initial, element.

Once again, because we saw that falsehood is the nullary disjunction in the proof-theoretic formulation, we should expect this analogy to carry over to the order-theoretic formulation. Indeed, the least element is the least upper bound of the empty set.

4.5 Order-theoretic IPL as lattice

As seen thus far, the order-theoretic formulation of IPL gives rise to a *lattice* as it establishes a preorder with finite meets and joins. The definition of a lattice assumed in this course may deviate from the one typically found in the literature, which usually considers a lattice to be a partial order with finite meets and joins. In this course, we deliberately ignore the property of antisymmetry. If we were to impose the property of antisymmetry on the order defined by entailment, then we would need to introduce equivalence classes of propositions, which requires associativity. As we will see later in this course, the axiom of univalence provides an elegant way of dealing with equivalence of propositions.

4.6 Implication as exponential

The elimination rule for implication (along with reflexivity of entailment) ensures that $A\ \mathsf{true}, A \supset B\ \mathsf{true} \vdash B\ \mathsf{true}$. For the order-theoretic formulation to be complete,

we include the rule

$$\frac{}{A \wedge (A \supset B) \leq B}$$

The introduction rule for implication ensures that $C\ \mathsf{true} \vdash A \supset B\ \mathsf{true}$ if $A\ \mathsf{true}, C\ \mathsf{true} \vdash B\ \mathsf{true}$. Once again, so that the order-theoretic formulation is complete, we have

$$\frac{A \wedge C \leq B}{C \leq A \supset B},$$

Taken together, these rules show that $A \supset B$ is the exponential of A and B.

As we have seen previously, the order-theoretic formulation of IPL gives rise to a lattice. Now we have just seen that it also supports exponentials. As a result, the order-theoretic formulation of IPL gives rise to a *Heyting algebra*. A Heyting algebra is a lattice with exponentials. As we will see later in this course, the notion of a Heyting algebra is fundamental in proving completeness of IPL. The proof also relies on the notion of a *complement* in a lattice. The complement $\overline{A}$ of A in a lattice is such that

1. $\top \leq \overline{A} \vee A$;
2. $\overline{A} \wedge A \leq \bot$.

It follows that a complement, if present, is a suitable notion of negation, but negation, defined via the exponential, is not necessarily a complement.

References

[1] Institute for Advanced Study. *Homotopy Type Theory: Univalent Foundations of Mathematics*. The Univalent Foundations Program, 2013. `http://homotopytypetheory.org/book/`.

[2] Robert Harper. Extensionality, intensionality, and Brouwer's dictum. `http://existentialtype.wordpress.com/2012/08/11/extensionality-intensionality-and-brouwers-dictum/`, August 2012.

[3] Robert Harper. Constructive mathematics is not metamathematics. `http://existentialtype.wordpress.com/2013/07/10/constructive-mathematics-is-not-meta-mathematics/`, July 2013.

[4] Frank Pfenning. Lecture notes on harmony. `http://www.cs.cmu.edu/~fp/courses/15317-f09/lectures/03-harmony.pdf`, September 2009.

[5] Frank Pfenning. Lecture notes on natural deduction. `http://www.cs.cmu.edu/~fp/courses/15317-f09/lectures/02-natded.pdf`, August 2009.

15-819 Homotopy Type Theory
Week 2 Lecture Notes

Clive Newstead and Enoch Cheung

9/16/2013 and 9/18/2013

Foreword

These will undergo substantial revision and expansion in the coming week.

Recall from last time that we can think of the judgement $A\ \mathsf{true}$ as meaning 'A has a proof' and of $A\ \mathsf{false}$ as 'A has a refutation', or equivalently '$\neg A$ has a proof'. These atomic judgements give rise to hypothetical judgements of the form

$$A_1\ \mathsf{true}, A_2\ \mathsf{true}, \ldots, A_n\ \mathsf{true} \vdash A\ \mathsf{true}$$

The inference rules of intuitionistic propositional logic then give rise to the structure of a Heyting algebra, called the *Lindenbaum algebra*.

1 Lindenbaum algebras

Recall that IPL has the structure of a preorder, where we declare $A \leq B$ if and only if $A\ \mathsf{true} \vdash B\ \mathsf{true}$. Let T be some theory in intuitionistic propositional logic and define a relation $\simeq$ on the propositions in T by

$$A \simeq B \quad \text{if and only if} \quad A \leq B \text{ and } B \leq A$$

The fact that $\simeq$ is an equivalence relation follows from the more general fact if $(P, \leq)$ is a preorder and a relation $\equiv$ is defined on P by declaring $p \equiv q$ if and only if $p \leq q$ and $q \leq p$, then $\equiv$ is an equivalence relation on P.

Definition. The *Lindenbaum algebra* of T is defined to be the collection of $\simeq$-equivalence classes of propositions in T. Write $A^* = [A]_\simeq$. The ordering on the Lindenbaum algebra is inherited from $\leq$.

Theorem. The judgement $\Gamma \vdash A\ \mathsf{true}$ holds if and only if $\Gamma^* \vdash A^*$ holds in every Heyting algebra.

Proof. Exercise. □

2 Decidability and stability

Definition. $A\ \mathsf{prop}$ is *decidable* if and only if $A \vee \neg A\ \mathsf{true}$.

Decidability is what separates constructuve logic from classical logic: in classical logic, every proposition is decidable (this is precisely the law of the excluded middle), but in constructive logic, this is not so.

A sensible first question to ask might be: 'do decidable propositions exist?' Fortunately, the answer is affirmative.

- $\top$ and $\bot$ are decidable propositions;
- We would expect $m =_{\mathbb{N}} n$ to be a decidable proposition, where $=_{\mathbb{N}}$ denotes equality on the natural numbers;
- We would *not* expect $x =_{\mathbb{R}} y$ to be a decidable proposition, where $=_{\mathbb{R}}$ denotes equality on the real numbers, because real numbers are not finite objects.

Definition. $A\ \mathsf{prop}$ is *stable* if and only if $(\neg\neg A) \supset A\ \mathsf{true}$.

Again, in classical logic, every proposition is stable; in fact, the proposition $(\neg\neg A) \supset A\ \mathsf{true}$ is often taken as an axiom of treatments of classical propositional logic! A natural question to ask now is 'do there exist unstable propositions?' Consider the following lemma.

Lemma. $\neg\neg(A \vee \neg A)\ \mathsf{true}$

Proof. We must show $\neg(A \vee \neg A) \supset \bot\ \mathsf{true}$.

Suppose $A\ \mathsf{true}$. We then have

$$\dfrac{\dfrac{A\ \mathsf{true}}{A \vee \neg A\ \mathsf{true}}\ \vee I_1 \qquad \neg(A \vee \neg A)\ \mathsf{true}}{\bot}$$

So in fact $\neg A\ \mathsf{true}$. But then once again

$$\dfrac{\dfrac{\neg A\ \mathsf{true}}{A \vee \neg A\ \mathsf{true}}\ \vee I_2 \qquad \neg(A \vee \neg A\ \mathsf{true})}{\bot}$$

Hence

$$\frac{\neg(A \vee \neg A)\ \mathsf{true} \vdash \bot}{\neg(A \vee \neg A) \supset \bot\ \mathsf{true}}\ \supset I$$

□

We can think of this lemma as saying that 'the law of the excluded middle is not refutable'. Presuming that there exist undecidable propositions, we obtain the following corollary.

Corollary. In intuitionistic propositional logic, not every proposition is stable.

3 Disjunction property

A theory T has the *disjunction property (DP)* if $T \vdash A \vee B$ implies $T \vdash A$ or $T \vdash B$.

Theorem. In IPL, if $\varnothing \vdash A \vee B\ \mathsf{true}$ then $\varnothing \vdash A\ \mathsf{true}$ or $\varnothing \vdash B\ \mathsf{true}$.

Naïve attempt at proof. The idea is to perform induction on all possible derivations ∇ of $\varnothing \vdash A \vee B\ \mathsf{true}$, with the hope that somewhere along the line we'll find a derivation of $A\ \mathsf{true}$ or of $B\ \mathsf{true}$. Our induction hypothesis is that inside ∇ is enough information to deduce either $\varnothing \vdash A\ \mathsf{true}$ or $\varnothing \vdash B\ \mathsf{true}$.

Since $\varnothing \vdash A \vee B\ \mathsf{true}$ cannot be obtained by assumption or from the rules, $\wedge I$, $\supset I$ or $\top I$, we need only consider $\vee I_1$, $\vee I_2$ and the elimination rules.

If $\varnothing \vdash A \vee B\ \mathsf{true}$ is obtained from $\vee I_1$ then

$$\frac{\dfrac{\nabla}{A\ \mathsf{true}}}{\varnothing \vdash A \vee B\ \mathsf{true}}\ \vee I_1$$

so there is a derivation ∇ of $A\ \mathsf{true}$ and we're done. Likewise if $\varnothing \vdash A \vee B\ \mathsf{true}$ is obtained from $\vee I_2$ then there is a derivation of $B\ \mathsf{true}$.

If $\varnothing \vdash A \vee B\ \mathsf{true}$ is obtained from $\supset E$ then the deduction takes the form

$$\frac{\dfrac{\nabla_1}{\varnothing \vdash C \supset (A \vee B)\ \mathsf{true}} \quad \dfrac{\nabla_2}{\varnothing \vdash C\ \mathsf{true}}}{\varnothing \vdash A \vee B\ \mathsf{true}}\ \supset E$$

We (dubiously[1]) assume that $\vdash C \supset (A \vee B)\,\mathsf{true}$ must have been derived in some way from $C\ \mathsf{true} \vdash (A \vee B)\ \mathsf{true}$. Suppose that this happens and that ∇_1' is a deduction

[1] In fact, this 'dubious' assumption is true in constructive logic.

of $C\,\mathsf{true} \vdash (A \vee B)\,\mathsf{true}$. We can then 'substitute' ∇_2 for all the occurrences of the assumption $C\,\mathsf{true}$ appearing in ∇_1' to obtain a smaller derivation ∇_3 of $\varnothing \vdash A \vee B\,\mathsf{true}$. Our induction hypothesis then gives us that inside ∇_3 is enough information to deduce $\varnothing \vdash A\,\mathsf{true}$ or $\varnothing \vdash B\,\mathsf{true}$.

A similar approach works (we hope) for $\wedge E$, $\vee E$, and $\bot E$, thus giving the result. □

4 Admissible properties

The sketch proof of the previous theorem relied on transitivity of $\vdash$; namely, that the following rule is true:

$$\frac{\Gamma, A\,\mathsf{true} \vdash B\,\mathsf{true} \quad \Gamma \vdash A\,\mathsf{true}}{\Gamma \vdash B\,\mathsf{true}}\ \mathsf{T}$$

This leads us naturally into a discussion of the structural properties of $\vdash$.

Definition. A deduction rule is *admissible* (in IPL) if nothing changes when it is added to the existing rules of IPL.

To be clear about which logical system we use, we may write $\vdash_{\mathrm{IPL}}$ to denote deduction in IPL rather than in some new logical system.

The goal now is to prove that the structural rules for entailment (reflexivity, transitivity, weakening, contraction, exchange) are admissible.

Theorem. The structural properties of $\vdash_{\mathrm{IPL}}$ are admissible.

Proof. R, C, X: Reflexivity, contraction and exchange are all primitive notions, in that they follow instantly. For instance:

$$\cfrac{\cfrac{\Gamma \vdash A\,\mathsf{true}}{\Gamma \vdash A \wedge A\,\mathsf{true}}\ \wedge I}{\Gamma \vdash A\,\mathsf{true}}\ \wedge E_1$$

so if we were to introduce

$$\frac{\Gamma \vdash A\,\mathsf{true}}{\Gamma \vdash A\,\mathsf{true}}\ \mathsf{R}$$

as a new rule, then nothing would change. (Likewise for contraction and exchange.)

W: For weakening we use the fact that the structural rules are *polymorphic* in Γ. We can thus prove that weakening is admissible by induction: if the following rules are admissible

$$\frac{\Gamma \vdash B_1 \ \mathsf{true}}{\Gamma, A \ \mathsf{true} \vdash B_1 \ \mathsf{true}} \qquad \text{and} \qquad \frac{\Gamma \vdash B_2 \ \mathsf{true}}{\Gamma, A \ \mathsf{true} \vdash B_2 \ \mathsf{true}}$$

then we obtain

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash B_1 \wedge B_2 \ \mathsf{true}}{\Gamma \vdash B_1 \ \mathsf{true}} \wedge E_1}{\Gamma, A \ \mathsf{true} \vdash B_1 \ \mathsf{true}} \ \mathrm{Ind} \qquad \dfrac{\dfrac{\Gamma \vdash B_1 \wedge B_2 \ \mathsf{true}}{\Gamma \vdash B_2 \ \mathsf{true}} \wedge E_2}{\Gamma, A \ \mathsf{true} \vdash B_2 \ \mathsf{true}} \ \mathrm{Ind}}{\Gamma, A \ \mathsf{true} \vdash B_1 \wedge B_2 \ \mathsf{true}} \wedge I$$

Likewise for the other introduction rules.

T: The admissibility of transitivity is left as an exercise. □

5 Proof Terms

We wish to study propositions along with their proof as mathematical objects. In the type theoretic framework, we can use the notation $M : A$ where A is a proposition and M is a proof of A. We will see that this corresponds to the category theoretic notion of a mapping $M : A \to B$. Another important notion is the identity of proofs, which will be denoted $M \equiv N : A$ where M, N are equivalent proofs of A. This will correspond in the category theoretic contex to two maps from A to B being equal $M = N : A \to B$.

5.1 Proof Terms as Variables

We can combine the idea of keeping track of proofs with our previous notion of entailment. If $A_1, \ldots, A_n$ entails A, meaning that $A_1, \ldots, A_n \vdash A$, there will be a proof M of A that uses the propositions $A_1, \ldots, A_n$. Thus, we will write

$$x_1 : A_1, \ldots, x_n : A_n \vdash M : A$$

where each $x_i : A_i$ is a proof term. We can think of the proof terms $x_1, \ldots, x_n$ as hypothesise for the proof, but what we really want is for them to behave as variables. M then uses the variables $x_1, \ldots, x_n$ to prove A, so M would encapsulate the grammar a proof that uses variables $x_1, \ldots, x_n$.

Instead of proving a proposition A from nothing, most of the time A will rely on other propositions $A_1, \ldots, A_n$.

5.2 Structural Properties of Entailment with Proof Terms

Now that we have proof terms, we can see how they act as variables by examining their interaction with the structural properties of entailment. We will also keep track of other assumptions/context Γ, Γ' to demonstrate that the structural properties will hold in the presence of assumptions.

Reflexivity / Variables Rule Reflexivity tells us that A should entail A, so now that we have a variable $x : A$ that proves A, the variable should be carried through. We can think of this as the variables rule.

$$\frac{}{\Gamma, x : A, \Gamma' \vdash x : A}\ \mathsf{R/V}$$

Transitivity / Substitution Transitivity tells us that if A is true and B follows from A, then B is true. In terms of proofs, if we have a proof $M : A$ and a proof $N : B$ which uses a variable x that is supposed to prove A, then we can substitute the proof $M : A$ into $N : B$ to prove B. Since we are substituting M into x inside N, we denote this substitution $[M/x]N : B$.

$$\frac{\Gamma, x : A, \Gamma' \vdash N : B \quad \Gamma \vdash M : A}{\Gamma, \Gamma' \vdash [M/x]N : B}\ \mathsf{T/S}$$

Weakening

$$\frac{\Gamma \vdash M : A}{\Gamma, \Gamma' \vdash M : A}\ \mathsf{W}$$

Contraction If $N : B$ follows from A using two different proofs $x : A, y : A$ for A, we can just pick one $z = x$ or $z = y$ as the proof of $z : A$ and use it in the instances of variables x, y in $N : B$

$$\frac{\Gamma, x : A, y : A, \Gamma' \vdash N : B}{\Gamma, z : A, \Gamma' \vdash [z, z/x, y]N : B}\ \mathsf{C}$$

Exchange

$$\frac{\Gamma, x : A, y : B, \Gamma' \vdash N : C}{\Gamma, y : B, x : A, \Gamma' \vdash N : C}\ \mathsf{X}$$

5.3 Negative Fragment of IPL with Proof Terms

We want to look at what happens to the Negative Fragment of IPL when we consider proof terms. Here are the important ones:

Truth Introduction Truth is trivially true, so we have

$$\frac{}{\Gamma \vdash \langle\,\rangle : \top}\ \top I$$

Conjunction Introduction We combine the proofs $M : A$ and $N : B$ into $\langle M, N\rangle : A \wedge B$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N\rangle : A \wedge B}\ \wedge I$$

Conjunction Elimination We can recover from a proof $M : A \wedge B$ proofs of A and B

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathrm{fst}(M) : A}\ \wedge E_1 \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \mathrm{snd}(M) : B}\ \wedge E_2$$

Implication Introduction If we have a proof $M : B$ that uses $x : A$ as a variable, then we can consider $\lambda x.M$ as a function that maps x a variable to a proof of B that uses x, which proves that $A \supset B$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \supset B}\ \supset I$$

Implication Elimination By applying an actual proof $N : A$ to the function described above, we obtain a proof $M(N) : B$

$$\frac{\Gamma \vdash M : A \supset B \quad \Gamma \vdash N : A}{\Gamma \vdash M(N) : B}\ \supset E$$

6 Identity of Proofs

6.1 Definitional Equality

We want to think about when two proofs $M : A$ and $M' : A$ are the same. We will introduce an equivalence relation called *definitional equality* that respects the

proof rules, denoted $M \equiv M' : A$. We want definitional equality $\equiv$ to be the least congruence containing (closed under) the β rules. We will define what this means:

A *congruence* is an equivalence relation that respects our operators. Being an equivalence relation that it is reflexive ($M \equiv M : A$), symmetric ($M \equiv N : A$ implies that $N \equiv M : A$), and transitive ($M \equiv N : A$ and $N \equiv M' : A$ implies that $M \equiv M' : A$).

For the equivalence relation to respect our operators basically means that if $M \equiv M' : A$, then that their image under any operator should be equivalent. In other words, we should be able to replace M with M' everywhere. For example

$$\frac{\Gamma \vdash M \equiv M' : A \wedge B}{\Gamma \vdash \mathrm{fst}(M) \equiv \mathrm{fst}(M') : A}$$

There can be many congruences that contains the β rules. Given two congruences $\equiv$ and $\equiv'$, we say $\equiv$ is finer than $\equiv'$ if $M \equiv' N : A$ implies that $M \equiv N : A$. The least congruence that contains the proof rules is the finest congruence that contains the β rules. We will define the β rules in the next section.

We will give a more explicit definition to definitional equality later.

6.2 Gentzen's Inversion Principle

Gentzen's Inversion Principle captures the idea that "elim is post-inverse to intro," which is the informal notion that the elimination rules should cancel the introduction rules, modulo definitional equality. The following are the β rules for the negative fragment of IPL:

Conjunction When we introduce a conjunction, we combine proofs $M : A$ and $N : B$ to produce a proof $\langle M, N \rangle : A \wedge B$. When we eliminate a conjunction, we retrieve $M : A$ or $N : B$. We do not want this process to alter our original M or N

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \mathrm{fst}(\langle M, N \rangle) \equiv M : A}\ \beta\wedge_1$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \mathrm{snd}(\langle M, N \rangle) \equiv N : B}\ \beta\wedge_2$$

Implication When we introduce an implication, we convert a proof $M : B$ which uses some variable $x : A$ to a function which uses a variable x to produce a proof of

B. When we eliminate implication, we apply the proof of $A \supset B$ to $N : A$ to produce a proof of B.

$$\frac{\Gamma, x : A \vdash M : B \quad \Gamma \vdash N : A}{\Gamma \vdash (\lambda x.M)(N) \equiv [N/x]M : B}\ \beta\supset$$

6.3 Gentzen's Unicity Principle

Gentzen's Unicity Princples on the other hand captures the idea that "intro is post-inverse to elim." Another way to think about it is that there should be only one way modulo definitional equivalence to prove something, which is the way we have described. They are the η rules, which are the following

Truth

$$\frac{\Gamma \vdash M : \top}{\Gamma \vdash M \equiv \langle\ \rangle : \top}\ \eta\top$$

Conjunction

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash M \equiv \langle \mathrm{fst}(M), \mathrm{snd}(M) \rangle : A \wedge B}\ \eta\wedge$$

Implication

$$\frac{M : A \supset B}{\Gamma \vdash M \equiv \lambda x.Mx : A \supset B}\ \eta\supset$$

7 Proposition as Types

There is a correspondence between propositions and types:

Propositions	Types
$\top$	1
$A \wedge B$	$A \times B$
$A \supset B$	function $A \to B$ or B^A
$\bot$	0
$A \vee B$	$A + B$

For now, note that meets like $\top$ and $A \wedge B$ correspond to products like 1 and $A \times B$, and joins like $\bot$ and $A \vee B$ correspond to coproducts like 0 and $A + B$. This correspondence should become more apparent as we go along. We will now introduce the objects on the right column.

8 Category Theoretic Approach

In a Heyting Algebra, we have a preorder $A \leq B$ when A implies B. However, we now wish to keep track of proofs, so if M is a proof from A to B, we want to think of it as a map $M : A \to B$.

Identity There should be an identity map

$$\mathrm{id} : A \to A$$

Composition We should be able to compose maps

$$\frac{f : A \to B \quad g : B \to C}{g \circ f : A \to C}$$

Coherence Conditions The identity map and composition of maps should behave like functions

$$\begin{aligned} \mathrm{id}_B \circ f &= f : A \to B \\ f \circ \mathrm{id}_A &= f : A \to B \\ h \circ (g \circ f) &= (h \circ g) \circ f : A \to D \end{aligned}$$

Now we can think about objects in the category that corresponds to propostions given in the correspondence.

Terminal Object 1 is the terminal object, also called the final object, which corresponds to $\top$. For any object A there is a unique map $A \to 1$. This corresponds to $\top$ being the the greatest object in a Heyting Algebra, meaning that for all A, $A \leq 1$.

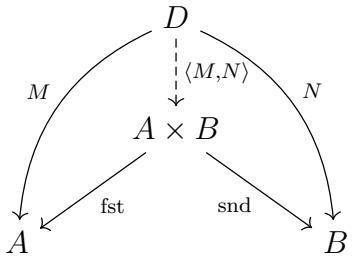
Existence:

$$\langle\ \rangle : A \to 1$$

Uniqueness:

$$\frac{M : A \to 1}{M = \langle\ \rangle : A \to 1}\ \eta\top$$

Product For any objects A and B there is an object $C = A \times B$ that is the *product* of A and B, which corresponds to the join $A \wedge B$. The product $A \times B$ has the following universal property:



where the diagram commutes.

First, the existence condition means that there are maps

$$\begin{aligned} \text{fst} &: A \times B \to A \\ \text{snd} &: A \times B \to B \end{aligned}$$

The universal property says that for every object D such that $M : D \to A$ and $N : D \to B$, there exists a unique map $\langle M, N\rangle : D \to A \times B$ such that

$$\frac{M : D \to A \quad N : D \to B}{\langle M, N\rangle : D \to A \times B}$$

and the diagram communtes meaning

$$\begin{aligned} \text{fst} \circ \langle M, N\rangle &= M : D \to A \qquad (\beta\times_1) \\ \text{snd} \circ \langle M, N\rangle &= N : D \to B \qquad (\beta\times_2) \end{aligned}$$

Furthermore, the map $\langle M, N\rangle : D \to A \times B$ is unique in the sense that

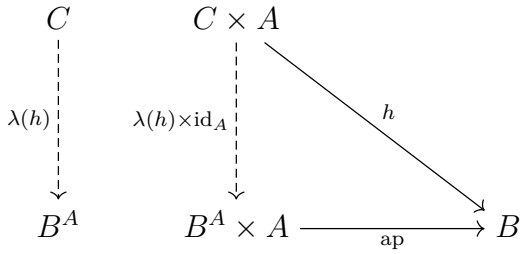
$$\frac{P : D \to A \times B \quad \text{fst} \circ P = M : D \to A \quad \text{snd} \circ P = N : D \to B}{P = \langle M, N\rangle : D \to A \times B}\ \eta\times$$

so in other words $\langle \text{fst} \circ P, \text{snd} \circ P\rangle = P$.

Another way to say the above is

$$\begin{aligned} \langle \text{fst}, \text{snd}\rangle &= \text{id} \\ \langle M, N\rangle \circ P &= \langle M \circ P, N \circ P\rangle \end{aligned}$$

Exponentials Given objects A and B, an exponential B^A (which corresponds to $A \supset B$) is an object with the following universal property:



such that the diagram commutes.

This means that there exists a map $\text{ap} : B^A \times A \to B$ (application map) that corresponds to implication elimination.

The universal property is that for all objects C that have a map $h : C \times A \to B$, there exists a unique map $\lambda(h) : C \to B^A$ such that

$$\text{ap} \circ (\lambda(h) \times \text{id}_A) = h : C \times A \to B$$

This means that the diagram commutes. Another way to express the induced map is $\lambda(h) \times \text{id}_A = \langle \lambda(h) \circ \text{fst}, \text{snd} \rangle$.

The map $\lambda(h) : C \to B^A$ is unique, meaning that

$$\frac{\text{ap} \circ (g \times \text{id}_\text{A}) = h : C \times A \to B}{g = \lambda(h) : C \to B^A}\ \eta$$

References

Week 3 Lecture Notes

Enoch Cheung and Clive Newstead

9/30/2013 and 10/2/2013

1 The β and η Rules

1.1 Gentzen's Inversion Principles (β rules)

Recall the β rules:

$$\begin{aligned}
&\wedge_1 : \mathsf{fst}\langle M, N\rangle \equiv M \\
&\wedge_2 : \mathsf{snd}\langle M, N\rangle \equiv N \\
&\supset_1 : (\lambda x.M)(N) \equiv [N/x]M \\
&\vee_1 : \mathsf{case}(\mathsf{inl}(M); x.P, y.Q) \equiv [M/x]P \\
&\vee_2 : \mathsf{case}(\mathsf{inr}(M); x.P, y.Q) \equiv [M/y]Q
\end{aligned}$$

The β rules can be expressed very compactly, and tells us that elimination rules should "cancel out" introduction rules. The notation used here is $\mathsf{inl}(x)$ (inject left) to mean using the proof x to prove A to prove $A \vee B$, and $\mathsf{inr}(x)$ (inject right) to mean using the proof x to prove B to prove $A \vee B$. $\mathsf{case}(x, y, z)$ means "if x, then y, else z." This also expresses the idea of dynamics of proofs, meaning that proofs can be viewed as programs.

1.2 Gentzen's Unicity Principles (η rules)

Recall the η rules we have given so far:

Truth

$$\frac{\Gamma \vdash M : \top}{\Gamma \vdash M \equiv \langle\,\rangle : \top}\ \eta\top$$

Conjunction

$$\frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash M \equiv \langle \mathsf{fst}(M), \mathsf{snd}(N) \rangle : A \wedge B}\ \eta\wedge$$

Implication

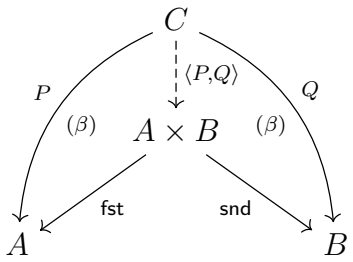
$$\frac{M : A \supset B}{\Gamma \vdash M \equiv \lambda x.Mx : A \supset B}\ \eta\supset$$

The η rules on the other hand takes a little bit more to write out, and expresses uniqueness (up to equivalence) of proofs of certain types.

The η conjunction rule can be expressed another way:

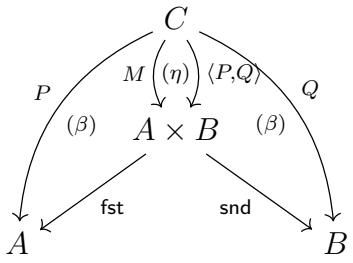
$$\frac{\Gamma \vdash M : A \wedge B \quad \Gamma \vdash \mathsf{fst}(M) \equiv P : A \quad \Gamma \vdash \mathsf{snd}(M) \equiv Q : B}{\Gamma \vdash \langle P, Q \rangle \equiv M : A \wedge B}\ \eta\wedge$$

This time we give equivalent proof terms P, Q to $\mathsf{fst}(M)$ and $\mathsf{snd}(N)$. This corresponds to the product diagram (where $A \wedge B$ corresponds to the product $A \times B$):



which says that given any object C with maps $P : C \to A$ and $Q : C \to B$, there exists a unique map $\langle P, Q \rangle : C \to A \times B$ such that the β rules make each cell commute, meaning $P \equiv \mathsf{fst}(\langle P, Q \rangle)$ and $Q \equiv \mathsf{snd}(\langle P, Q \rangle)$.

The η rule, on the other hand, gives uniqueness of the $\langle P, Q \rangle$ map, expressed as



where the η rule maks the center cell commute, meaning that given any map $M : C \to A \times B$ such that $\mathsf{fst} \circ M \equiv P$ and $\mathsf{snd} \circ M \equiv Q$, we have $M \equiv \langle P, Q \rangle$, expressing the uniqueness of the $\langle P, Q \rangle$ map.

While η rules gives the uniqueness of the product map, one can ask whether the product object $A \times B$ is unique. In a Heyting algebra, we can show that if $A \wedge' B$ has the same properties (being a greatest lower bound of A and B) as $A \wedge B$, then:

$$A \wedge B \leq A \wedge' B \qquad A \wedge B \geq A \wedge' B$$

If we think of them as objects, then we have maps $F : A \times B \to A \times' B$ and $G : A \times B \to A \times' B$ such that $F \circ G = \mathsf{id}$ and $G \circ F = \mathsf{id}$, which gives $A \times B \cong A \times' B$. With the Univalence axiom, we identify equivalence things as being equal, so we can say that $A \times B = A \times' B$.

1.3 η rule for Disjunction $\vee$

We wish to give the η rule for $\vee$, but if we were to attempt to naively define it as we did before for $M : A \vee B$, then we might force M to be a proof of A or B, because a proof of A also proves $A \vee B$, but forcing proofs of A or a proof of B to be identified with a proof of $A \vee B$ would not make sense.

We will take inspiration from Shannon expansions, specifically the concept of case analysing to give two different proofs. As a toy example, we consider a proof of $\top \vee \top$

$$\begin{aligned} \mathsf{inl}(\langle\,\rangle) &= \mathsf{true} \\ \mathsf{inr}(\langle\,\rangle) &= \mathsf{false} \\ \langle\,\rangle &: \top \vee \top \\ \mathsf{case}(M; P, Q) &= \text{“if } M \text{ then } P \text{ else } Q\text{”} : \top \vee \top \end{aligned}$$

where we look at the variable M and decide to use P or Q (here P, Q does not have an input because there is no data for proof of $\top$ anyways). The is an example of a Binary Decision Diagram (BDD), because we are making a decision when examining the variable M and branching to two cases.

In general, we can imagine a much bigger BDD which has many variables examined sequentially, and look at the Shannon expansion at some variable M in the middle. The idea here is that $M = \mathsf{true}$ and $M = \mathsf{false}$ will lead to two different subtrees. We write

$$[M/z]P \equiv \text{if } M \text{ then } [\mathsf{true}/z]P \text{ else } [\mathsf{false}/z]P$$

where when we look at the variable M which P uses, there are two cases: the case where M is true, which gives $[\mathsf{true}/z]P$ and the case where M is false which gives $[\mathsf{false}/z]P$. The point is that $[\mathsf{true}/z]P$ and $[\mathsf{false}/z]P$ do not have M as a variable, so M is fixed at a value.

Another notation for this is

$$P_z \equiv \text{if } z \text{ then } [\mathsf{true}/z]P \text{ else } [\mathsf{false}/z]P$$

To write the η rule for $\vee$, we will describe what happens when a proof $P : C$ uses a proof term $z : A \vee B$, meaning that the C follows from $A \vee B$. Now suppose we have a proof $M : A \vee B$, and we want to look at what happens when we make $P : C$ include $M : A \vee B$ by doing the substitution $[M/z]P : C$.

$$\frac{\Gamma \vdash M : A \vee B \quad \Gamma, z : A \vee B \vdash P : C}{\Gamma \vdash [M/z]P \equiv \mathsf{case}(M; x.[\mathsf{inl}(x)/z]P, y.[\mathsf{inr}(y)/z]P) : C}\ \eta\vee$$

This can be thought of as a "generalized Shannon expansion," where the Shannon expansion can be recovered as a special case

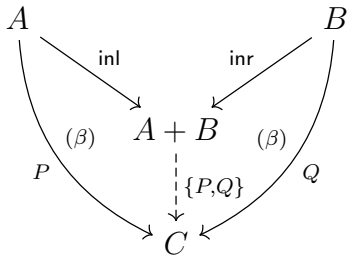
$$M \equiv \mathsf{case}(M; x.\mathsf{inl}(x); y.\mathsf{inr}(y))$$

1.4 Coproduct

In the category theoretical view, the disjunction $A \vee B$ corresponds to the coproduct $A + B$, with $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ being the canonical injections.

To give some intuitions about the coproduct, if we were in the category of sets, we can think of $A + B = A \sqcup B = (\{0\} \times A) \cup (\{1\} \times B)$ as a disjoint union, then $\mathsf{inl}, \mathsf{inr}$ would be the canonical embeddings $\mathsf{inl} : a \mapsto (0, a)$ and $\mathsf{inr} : b \mapsto (1, b)$. If A, B are already disjoint, then we can let $A + B = A \cup B$ and $\mathsf{inl}, \mathsf{inr}$ would be the inclusion maps $\mathsf{inl} : a \mapsto a$ and $\mathsf{inr} : b \mapsto b$.

The β rule for $\vee$ gives the following commutative diagram:



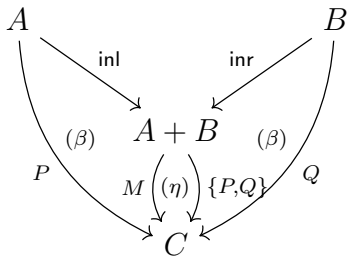
Where given any object C with maps $P : A \to C$ and $Q : B \to C$, there exists a unique map $\{P, Q\} : A + B \to C$ that is the copair of maps P, Q, which in our context corresponds to

$$\{P, Q\} \approx \mathsf{case}(-; x.P, y.Q)$$

The β rule makes the diagram commute, meaning that the composition of maps $P \equiv \{P, Q\} \circ \mathsf{inl}$ and $Q \equiv \{P, Q\} \circ \mathsf{inr}$. Written another way:

$$\mathsf{case}(\mathsf{inl}(-); x.P, y.Q) \equiv [-/x]P$$
$$\mathsf{case}(\mathsf{inr}(-); x.P, y.Q) \equiv [-/x]Q$$

The η rule expresses uniqueness, which is demonstrated by the following diagram



where given a map $M : A + B \to C$ such that $M \circ \mathsf{inl} \equiv P$ and $M \circ \mathsf{inr} \equiv Q$, the map is in fact equivalent to $M \equiv \{P, Q\}$, so the η rule makes the center cell commute.

Just as we have done for $\eta\wedge$, we can rewrite the $\eta\vee$ rule by explicitly naming $P : Q \to C$ and $Q : B \to C$ as follows

$$\frac{\begin{array}{c}\Gamma, z : A + B \vdash M : C \\ \Gamma, x : A \vdash [\mathsf{inl}(x)/z]M \equiv P : C \\ \Gamma, y : B \vdash [\mathsf{inr}(y)/z]M \equiv Q : C\end{array}}{\Gamma, z : A + B \vdash M \equiv \mathsf{case}(z; x.P, y.Q) : C}\ \eta\vee$$

1.5 Definitional equality vs. Propositional equality

Our different treatments of β rules and η rules above suggests that there is something fundamentally different between equivalence given by β rules and equivalence given by η rules. Indeed, there is a distinction which we will make more clear later. For now, note that

β rules	Analytical ("self-evident")	Definitional equality
η rules	Synthetic ("require proof")	Propositional equality

The β rules can be thought of as self-evident, or analytical, because it just says that our notation such as $\mathsf{fst}, \mathsf{snd}, \langle -, - \rangle, \mathsf{inl}, \mathsf{inr}, \mathsf{case}$ should behave the way we expect them to. On the other hand, the η rules are not so obvious, and expresses the equivalence of two things that behaves the same way, so they are synthetic, or requires proof instead of being self-evident.

The notion of equality produced by β rules is called definitional equality, or judgemental equality, which is more basic. The notion of equality produced by η rules is called propositional equality, which has to be expressed by a type (so it is typical).

2 Natural numbers

We'd like to capture the idea of definition by recursion. We will do so in two ways. First we will implement the natural numbers syntactically as a type, denoted Nat—it is a ubiquitous example of an inductively defined type. Then we will implement the natural numbers in a category theoretic context, as a so-called natural numbers object (NNO), denoted $\mathbb{N}$.

2.1 Syntactic definition: Nat

The type Nat has two introduction rules:

$$\frac{}{\Gamma \vdash 0 : \mathsf{Nat}}\ \mathsf{Nat}\text{-}I_0, \qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash s(M) : \mathsf{Nat}}\ \mathsf{Nat}\text{-}I_s$$

and one elimination rule, which can be thought of as a `for` loop or a recursion:

$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma \vdash P : A \quad \Gamma, x : A \vdash Q : A}{\Gamma \vdash \mathsf{rec}(P, x.Q)(M) : A}\ \mathsf{Nat}\text{-}E$$

We call rec the *recursor*.

We can think of 0 as being zero and s as being the successor operation, which takes a natural number n to its successor $n + 1$.

The β-rules for Nat are what they 'should be':

$$\frac{\Gamma \vdash P : A \quad \Gamma \vdash Q : A}{\Gamma \vdash \mathsf{rec}(P, Q)(0) \equiv P : A}\ \beta\text{-}\mathsf{Nat}_0$$

$$\frac{\Gamma \vdash P : A \quad \Gamma \vdash Q : A}{\Gamma \vdash \mathsf{rec}(P, Q)(s(M)) \equiv [\mathsf{rec}(P, Q)(M)/x]Q : A}\ \beta\text{-}\mathsf{Nat}_s$$

The η-rule for the NNO is somewhat ugly:

$$\frac{\Gamma, z : \mathsf{Nat} \vdash M : A \quad \Gamma, z : \mathsf{Nat} \vdash [s(z)/z]M \equiv [M/x]Q \quad \Gamma \vdash [0/z]M \equiv P : A}{\Gamma, z : \mathsf{Nat} \vdash M \equiv \mathsf{rec}(P, Q)(z) : A}\ \eta\text{-}\mathsf{Nat}$$

It says that 'if something behaves like the recursor, then it is the recursor'.

Given $n \in \mathbb{N}$, define the numeral $\bar{n} = \underbrace{s(s(\cdots s}_{n \text{ times}}(0)\cdots))$. With a slight abuse of notation, the β then tells us that

$$\mathsf{rec}(P, Q)(\bar{n}) \equiv \underbrace{Q(Q(\cdots Q}_{n \text{ times}}(P)\cdots))$$

That is, $\mathsf{rec}(P, Q)(0) \equiv P$ and $\mathsf{rec}(P, Q)(\overline{n+1}) \equiv Q(\mathsf{rec}(P, Q)(\bar{n}))$. This is precisely a definition by recursion.

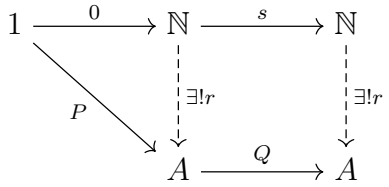
A special case of this is when $P = 0$ and Q is the successor operation. Then

$$z : \mathsf{Nat} \vdash \mathsf{rec}(0, s.s(y))(z) \equiv z : \mathsf{Nat}$$

This is what we'd expect: if you apply the successor operation to 0 n times then what you obtain is n.

2.2 Category theoretic definition: NNO

Fix a category $\mathcal{C}$ and suppose that $\mathcal{C}$ has a terminal object 1. A natural numbers object in $\mathcal{C}$ is an object $\mathbb{N}$ equipped with arrows $0 : 1 \to \mathcal{C}$ and $s : \mathcal{C} \to \mathcal{C}$ satisfying the following univeral property:



That is, given any morphism $P : 1 \to A$ and $Q : A \to A$ there exists a unique morphism $r = \mathsf{rec}(P, Q) : \mathbb{N} \to A$ such that

$$\mathsf{rec}(P, Q) \circ 0 = P \quad \text{and} \quad \mathsf{rec}(P, Q) \circ s = Q \circ \mathsf{rec}(P, Q)$$

These two equations correspond precisely with the β rules for Nat.

The η rule corresponds with the uniqueness: if $M : \mathbb{N} \to A$ satisfies $M \circ s = Q \circ M$ and $M \circ 0 = P$ then $M = \mathsf{rec}(P, Q)$.

Concrete example

In the category of sets, take $\mathbb{N}$ to be the set of natural numbers. The terminal object is any singleton $\{*\}$, and we can define $0 : \{*\} \to \mathbb{N}$ by $0(*) = 0 \in \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$ by $s(n) = n + 1$. Then the triple $(\mathbb{N}, 0, s)$ defines a natural numbers object: if $P \in A$ and $Q : A \to A$ then we can define $\mathsf{rec}(P, Q) : \mathbb{N} \to A$ by

$$\mathsf{rec}(P, Q)(0) = P \quad \text{and} \quad \mathsf{rec}(P, Q)(n + 1) = Q(\mathsf{rec}(P, Q)(n))$$

It is then clear that the above diagram commutes, and we can prove that $\mathsf{rec}(P, Q)$ is the unique such function by induction on its argument.

NNO as an initial algebra

There is an equivalent definition of a natural numbers object as an *initial algebra*.

Given an endofunctor (i.e. a functor F from a category $\mathcal{C}$ to itself), an *F-algebra* is a pair (A, α), where A is an object in the category and $\alpha : F(A) \to A$ is a morphism.

A *homomorphism of F-algebras* $f : (A, \alpha) \to (B, \beta)$ is a map $f : A \to B$ making the following square commute:

$$\begin{array}{ccc} F(A) & \xrightarrow{F(f)} & F(B) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{f} & B \end{array}$$

That is, f respects α and β in the only way it can.

An *initial F-algebra* is an F-algebra (I, ι) such that given any other F-algebra (A, α) there exists a unique F-algebra homomorphism $(I, \iota) \to (A, \alpha)$.

With these definitions in mind, a natural numbers object is precisely an initial F-algebra, where F is the functor $1 + (-)$.

To see how this functor acts on morphisms, consider the more general scenario of having morphisms $f : A \to A'$ and $g : B \to B'$. Then we have morphisms

$$\mathsf{inl} \circ f : A \to A' + B' \quad \text{and} \quad \mathsf{inr} \circ g : B \to A' + B'$$

Then the universal property of the coproduct gives rise to a map

$$f + g = \{\mathsf{inl} \circ f, \mathsf{inr} \circ g\} : A + B \to A' + B'$$

What this means more concretely is as follows. A natural numbers object is an object $\mathbb{N}$ equipped with a morphism $\{0, s\} : 1 + \mathbb{N} \to \mathbb{N}$ such that if $\{P, Q\} : 1 + A \to A$ is another morphism then there is a unique morphism $\mathsf{rec}(P, Q) : \mathbb{N} \to A$ such that $\{P, Q\} \circ (1 + \mathsf{rec}(P, Q)) = \mathsf{rec}(P, Q) \circ \{0, s\}$.

3 Intensional and extensional equality

We can implement addition by

$$p = \lambda x \lambda y \, \mathsf{rec}(x, z.s(z))(y)$$

Given numerals $\bar{m}$ and $\bar{n}$ it is clear that $p\,\bar{m}\,\bar{n} = \overline{m+n}$, so this definition does implement $+$.

We could have recursed on x instead of y. Indeed, we can define $q = \lambda x \lambda y \, pyx$.

Again we can prove that $q\,\bar{m}\,\bar{n} = \overline{m+n}$, so q is another implementation of addition.

Despite this fact, we will not in general be able to prove

$$x : \mathsf{Nat}, y : \mathsf{Nat} \vdash pxy \equiv qxy$$

This seems odd: for every $m, n \in \mathbb{N}$ (in the 'real world') we can prove that $p\,\bar{m}\,\bar{n} = q\,\bar{m}\,\bar{n}$. If we had a principle of induction then we'd be able to deduce that $pxy = qxy$ generically. However, we have no such principle!

Morally this should not be the case: that is, p and q are not *definitionally equal*. This illustrates the distinction between *intensional equality* (a.k.a. *definitional equality*) and *extensional equality*. This distinction is very important in computer science and philosophy: it captures the idea of two programmes having the same input–output behaviour but different algorithms.

Extensional equality. We can think of the *extension* of a function as being its graph, i.e. a set of ordered pairs of the form $(\mathsf{input}, \mathsf{output})$. Two programmes may have the same input/output behaviour without being the same programme. In Frege's terminology, two types are extensionally equal if they have the same *reference*.

We cannot expect extensional equality to be computable; for instance, extensional equality of elements of type $(\mathbb{N} \to \mathbb{N}) \to (\mathbb{N} \to \mathbb{N})$ already has high quantifier complexity.

Intensional equality. We can think of the *intension* of a function as being its description, or an algorithm that computes the function. Thus two functions that are intensionally equal must be extensionally equal, but the converse is not true. Intensional equality is synthetic. In Frege's terminology, two types are intensionally equal if they have the same *sense*.

3.1 Equality in type theory

Recall Martin–Löf's distinction between judgements and propositions. With this in mind:

- Intensional equality is an inductively defined judgement;
- Extensional equality is a proposition: it may be subject to judgement.

For example, the following is a proposition:

$$pxy =_{\mathsf{Nat}} qxy$$

It requires proof. We will attempt to develop a way of saying that, to prove this, it is sufficient to prove for each $m, n \in \mathbb{N}$ that $p\,\bar{m}\,\bar{n} = q\,\bar{m}\,\bar{n}$.

Under our propositions-as-types correspondence, we conclude that extensional equality 'is' a family of types. For instance,

$$x : \mathsf{Nat}, y : \mathsf{Nat} \vdash x =_{\mathsf{Nat}} y\ \mathsf{type} \tag{1}$$

We'll write the type $x =_{\mathsf{Nat}} y$ as $\mathsf{Id}_{\mathsf{Nat}}(x, y)$ to emphasise that we really want to think of it as a type and not a proposition.

Instantiating by substitution from 1 gives

$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash \mathsf{Id}_{\mathsf{Nat}}(M, N)\ \mathsf{type}}$$

But we needn't stop at Nat; we may replace it by an arbitrary type A (which may itself—usefully—be an identity type!). For instance, given $x : \mathsf{Nat}$ we may obtain a new type $\mathsf{Seq}(x)$, which can be thought of as the sequences of Nats of length x:

$$\frac{\Gamma \vdash x : \mathsf{Nat}}{\Gamma \vdash \mathsf{Seq}(x) :\ \mathsf{type}}$$

Observe the following fact: given $m, n \in \mathbb{N}$, it is true that

$$\mathsf{Seq}(p\,\bar{m}\,\bar{n}) \equiv \mathsf{Seq}(q\,\bar{m}\,\bar{n})$$

because $p\,\bar{m}\,\bar{n} \equiv q\,\bar{m}\,\bar{n}$. However we cannot generalise to

$$\mathsf{Seq}(pxy) \equiv \mathsf{Seq}(qyx)$$

because $\mathsf{Seq}(pxy)$ and $\mathsf{Seq}(qyx)$ are not definitionally equivalent. But they *are* related in some way. Later, we will come to define what we mean by 'related' here. A good guess might be along the lines of 'isomorphism', but this will turn out to be far too strong. What we need is some kind of 'equivalence'. This equivalence will tie itself to both the notion of a homotopy and that of a categorical equivalence.

4 Dependent types: setup

Dependent types are *families* of types. Atomic judgements are of the form

contexts / closed types:	$\Gamma\ \mathsf{ctx}$
	$\Gamma \equiv \Gamma'$
open types / families of types:	$\Gamma \vdash A\ \mathsf{type}$
	$\Gamma \vdash A \equiv A'$
elements of types:	$\Gamma \vdash M : A$
	$\Gamma \vdash M \equiv M' : A$

The symbol $\equiv$ denotes what we will interpret as *definitional equality*. We denote the *empty context* by $\cdot$ when we need to. The introduction rules for contexts are:

$$\frac{}{\cdot\ \mathsf{ctx}} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A\ \mathsf{ctx}}$$

Thus we have some notion of *dependence*; it allows us to make sense of expressions like $x : \mathsf{Nat}, y : \mathsf{Seq}(x) \vdash \cdots$, which was impossible before.

$$\frac{}{\cdot \equiv \cdot\ \mathsf{ctx}} \qquad \frac{\Gamma \equiv \Gamma' \quad \Gamma \vdash A \equiv A'}{\Gamma, x : A \equiv \Gamma', x : A'}$$

The following rule corresponds with reflexivity:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}$$

The following rules (one for each judgement J) correspond with weakening:

$$\frac{\Gamma, \Delta \vdash J \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A, \Delta \vdash J}$$

Exercise. What are the corresponding rules for exchange and contraction?

The following rule, called *substitution* or *instantiation*, corresponds with transitivity:

$$\frac{\Gamma, x : A, \Delta \vdash J \quad \Gamma \vdash M : A}{\Gamma[M/x]\Delta \vdash [M/x]J}$$

The following rules together are called *functionality*

$$\frac{\Gamma, x : A, \Delta \vdash N : B \quad \Gamma \vdash M \equiv M' : A}{\Gamma[M/x]\Delta \vdash [M/x]N \equiv [M'/x]N : [M/x]B}$$

$$\frac{\Gamma, x : A, \Delta \vdash B\ \mathsf{type} \quad \Gamma \vdash M \equiv M' : A}{\Gamma[M/x]\Delta \vdash [M/x]B \equiv [M'/x]B}$$

Finally, the following rules are *type equality*, which tell us that definitionally equal types classify the same things:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \equiv A'}{\Gamma \vdash M : A'} \qquad \frac{\Gamma \vdash M \equiv M' : A \quad \Gamma \vdash A \equiv A'}{\Gamma \vdash M \equiv M' : A}$$

Identity types

Given a type A and elements $M : A$ and $N : A$ we can form an *identity type* $\mathsf{Id}_A(M, N)$. The formation rule for Id is thus:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{Id}_A(M, N)}\ \mathsf{Id}\text{-}F$$

It will be useful in HoTT to consider the case when A is itself an identity type, i.e. we have the type

$$\mathsf{Id}_{\mathsf{Id}_A(A,B)}(\alpha, \beta)$$

This extends to any (finite) dimension.

We also have an Id-introduction rule, which tells us that any element M of a type A is in some way 'related' to itself. Formally:

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

Id-elimination will follow next week.

Week 4 Lecture Notes

Jason Koenig

10/7/2013 and 10/9/2013

1 Dependent Types

1.1 Structural Foundation

Dependent types are *families* of types. Recall from last week the structural setup, which differs from IPL in that we need to express *dependency* between terms and types in the context and the preceeding terms. Atomic judgements are of the form

contexts / closed types:	$\Gamma\ \mathsf{ctx}$
	$\Gamma \equiv \Gamma'$
open types / families of types:	$\Gamma \vdash A\ \mathsf{type}$
	$\Gamma \vdash A \equiv A'$
elements of types:	$\Gamma \vdash M : A$
	$\Gamma \vdash M \equiv M' : A$

The symbol $\equiv$ denotes what we will interpret as *definitional equality*. We denote the *empty context* by $\cdot$ when we need to. The introduction rules for contexts are:

$$\frac{}{\cdot\ \mathsf{ctx}} \qquad \frac{\Gamma\ \mathsf{ctx} \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A\ \mathsf{ctx}}$$

Thus we have some notion of *dependence*; it allows us to make sense of expressions like $x : \mathsf{Nat}, y : \mathsf{Seq}(x) \vdash \cdots$, which was impossible before.

$$\frac{}{\cdot \equiv \cdot\ \mathsf{ctx}} \qquad \frac{\Gamma \equiv \Gamma' \quad \Gamma \vdash A \equiv A'}{\Gamma, x : A \equiv \Gamma', x : A'}$$

The following rule corresponds with reflexivity:

$$\frac{}{\Gamma, x : A, \Delta \vdash x : A}$$

The following rules (one for each judgement J) correspond with weakening:

$$\frac{\Gamma, \Delta \vdash J \quad \Gamma \vdash A\ \mathsf{type}}{\Gamma, x : A, \Delta \vdash J}$$

Exercise. What are the corresponding rules for exchange and contraction?

The following rule, called *substitution* or *instantiation*, corresponds with transitivity:

$$\frac{\Gamma, x : A, \Delta \vdash J \quad \Gamma \vdash M : A}{\Gamma[M/x]\Delta \vdash [M/x]J}$$

The following rules together are called *functionality*

$$\frac{\Gamma, x : A, \Delta \vdash N : B \quad \Gamma \vdash M \equiv M' : A}{\Gamma[M/x]\Delta \vdash [M/x]N \equiv [M'/x]N : [M/x]B}$$

$$\frac{\Gamma, x : A, \Delta \vdash B\ \mathsf{type} \quad \Gamma \vdash M \equiv M' : A}{\Gamma[M/x]\Delta \vdash [M/x]B \equiv [M'/x]B}$$

Finally, the following rules are *type equality*, which tell us that definitionally equal types classify the same things:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash A \equiv A'}{\Gamma \vdash M : A'} \qquad \frac{\Gamma \vdash M \equiv M' : A \quad \Gamma \vdash A \equiv A'}{\Gamma \vdash M \equiv M' : A'}$$

1.2 Negative Dependent Types

In dependent type theory, the negative types need to be generalized to express dependency. The type $\top$ does not change. There are new forms for $\wedge$ and $\vee$, however:

IPL	Type		Dependent Type	Quantifier
$A \wedge B$	$A \times B$	$\to$	$\Sigma x : A.B$	$\exists x : A.B$
$A \supset B$	B^A or $A \to B$	$\to$	$\Pi x : A.B$	$\forall x : A.B$

Before we present the rules governing these types, it is good to get some intuition for what they represent. The dependent type $\Sigma x : A.B$, called the *sum*, *dependent product*, or *sigma-type*. Here unfortunately terminology becomes muddled, as Σ properly generalizes the product. The Σ type corresponds to (constructive) existential quantification, $\exists x : A.B$.

The type $\Pi x : A.B$ is called the *dependent product* (ambiguously with with above), or *pi-type*. This generalizes functions from $A \to B$ in that the type of the result of an application can depend on exactly which value the function is applied to. This corresponds to universal quantification $\forall x : A.B$.

Example: We can form complex types from these connectives and the families we have seen. For example, we can form the type $\Pi x : \mathsf{Nat}.\Sigma y : \mathsf{Nat}.\mathsf{Id}_{\mathsf{Nat}}(y, \mathrm{succ}(x))$. By the types-propositions correspondence, this is the proposition $\forall x : \mathsf{Nat}.\exists y : \mathsf{Nat}.y =_{\mathsf{Nat}} \mathrm{succ}(x)$, i.e. for every natural number, there is a successor. We can think of the as inhabited by proofs that the proposition is true. We can also think of elements of this type as terms which take a natural number x, and produce a term which is a proof that that particular natural has a successor.

Example: A less logical example is the type $\Pi x : \mathsf{Nat}.\mathrm{Seq}(x)$. This represents the type of functions which result in a length n sequence when applied to the value n, as one might do if "allocating" a sequence. This might be useful when programming, as the type encodes more information about a value than a non-dependent type could.

One important aspect of the quantifier view of these types is that they generalize classical logic in that they merge domains of quantification and proposition. In classical logic, and especially first order logic, these are completely distinct. With the identification of propositions and types, we can quantify not just over data (such as Nat), but also over proofs of propositions by letting A be a "proposition"-ish type like $\mathsf{Id}_{\mathsf{Nat}}(x, y)$.

This identification of dependent types with quantifiers only holds if you take the quantifiers as *constructive*. An element of type $\Sigma x : A.B$ consists of a *pair* $\langle x, y \rangle$ such that x is an witness of the existential, and y is a proof that that element means the condition. A constructive existential requires there actually to be an explicit element, which the proof must give on its own. This means one cannot use certain kinds of proofs from classical logic. Some of these proofs roughly show: $(\forall x.P) \supset \bot$, so they conclude that $\exists x.P$, but nowhere in the proof can we actually find such an x. Similarly, but less counterintuitively, a proof of an universal quantifier is a map from elements to proofs for those specific elements, i.e. a function. We cannot prove $\forall x.P$ by showing $(\exists x.P) \supset \bot$.

The idea with constructivity is then that if you have proved $(\forall x.P) \supset \bot$, then just let that be the proof. Constructivity amounts to a certain carefulness, where we avoid the classical convention that $\neg\exists \iff \forall$ and $\neg\forall \iff \exists$. By making this distinction, we get not only the ability to regard proofs as programs, but also the rich structure of HoTT, which would otherwise collapse.

1.2.1 Pi-types

The formation and introduction rules for pi-types are:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash \Pi x : A.B\ \mathsf{type}}\ \Pi\text{-F} \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : M : \Pi x : A.B}\ \Pi\text{-I}$$

Note that the introduction form is almost the same except that x is bound when forming the type B. The introduction rule is exactly the same as in IPL, except that the type B depends on x like M does. This dependency of B motivates the elimination rule:

$$\frac{\Gamma \vdash M : \Pi x : A.B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : [N/x]B}\ \Pi\text{-E}$$

which is the same as IPL except for the substitution of N, the actual argument, into the result type. Thus the result type can vary with the actual argument, which cannot happen in regular type theory.

We also have the (β) and (η) rules:

$$\begin{aligned} (\lambda x.M)N &\equiv [N/x]M \quad (\beta) \\ (\lambda x.Mx) &\equiv M \quad (\eta) \quad \text{(when } x \text{ not free in } M\text{)} \end{aligned}$$

1.2.2 Sigma-types

The formation and introduction rules for sigma-types are:

$$\frac{\Gamma \vdash A\ \mathsf{type} \quad \Gamma, x : A \vdash B\ \mathsf{type}}{\Gamma \vdash \Sigma x : A.B\ \mathsf{type}}\ \Sigma\text{-F} \qquad \frac{\Gamma \vdash M : A \quad \Gamma, x : A \vdash N : [M/x]B}{\Gamma \vdash \langle M, N \rangle : \Sigma x : A.B}\ \Sigma\text{-I}$$

The addition here is that the second component can depend on the first. The elimination rules are what one expects, with the addition of the dependence of the type of second component.

$$\frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \mathsf{fst}(M) : A}\ \Sigma\text{-E}_1 \qquad \frac{\Gamma \vdash M : \Sigma x : A.B}{\Gamma \vdash \mathsf{snd}(M) : [\mathsf{fst}(M)/x]B}\ \Sigma\text{-E}_2$$

The (β) and (η) rules are the familiar:

$$\begin{aligned} \mathsf{fst}(\langle M, N \rangle) &\equiv M \quad (\beta_1) \\ \mathsf{snd}(\langle M, N \rangle) &\equiv N \quad (\beta_2) \\ \langle \mathsf{fst}(M), \mathsf{snd}(M) \rangle &\equiv M \quad (\eta) \end{aligned}$$

1.3 Positive Dependent Types

With the positive types, the dependency does not affect the form of the types, but rather their elimination forms. The issue arises because we need a "join point" in the elimination. For example, C is the join point in the elimination of $\vee$ in IPL because it needs to hold in both cases:

$$\frac{\Gamma, x : A \vdash N : C \quad \Gamma, y : B \vdash P : C \quad \Gamma \vdash M : A + B}{\Gamma \vdash \mathsf{case}(M, x.N, y.P) : C} \vee\text{-E}_{\text{IPL}}$$

In IPL, there is no issue with not making C dependent because types are fixed: there is nothing that can vary about C in either branch. In dependent type theory, this is no longer the case.

1.3.1 Sum Types

To motivate why we need dependency in C, we consider the sum (i.e. disjunction). First we introduce the booleans, a simple case of the $(+)$ type. Let us define the boolean type 2 as $1 + 1$, where 1 is the unit type, and the values $\mathsf{tt} = \mathsf{inl}(\langle\rangle) : 2$ and $\mathsf{ff} = \mathsf{inr}(\langle\rangle) : 2$.

Consider the rather trivial proposition (written as a type) $\Pi x : 2.(\mathsf{Id}_2(x, \mathsf{ff})) + (\mathsf{Id}_2(x, \mathsf{tt}))$. In order to prove this, we need to perform a case analysis on x. In particular, the proposition we actually prove is different in each case:
1. False case: prove $(\mathsf{Id}_2(\mathsf{ff}, \mathsf{ff})) + (\mathsf{Id}_2(\mathsf{ff}, \mathsf{tt}))$ by left injection of reflexivity
2. True case: prove $(\mathsf{Id}_2(\mathsf{tt}, \mathsf{ff})) + (\mathsf{Id}_2(\mathsf{tt}, \mathsf{tt}))$ by right injection of reflexivity.
We see than if we case analyze on x, then the actual proposition we need to prove in each case is the goal we are trying to prove, specialized to the particular case we are in. The following rule codifies this:

$$\frac{\Gamma, x : A \vdash N : [\mathsf{inl}(x)/z]C \quad \Gamma, y : B \vdash P : [\mathsf{inr}(y)/z]C \quad \Gamma \vdash M : A + B \quad \Gamma, z : A + B \vdash C\ \mathsf{type}}{\Gamma \vdash \mathsf{case}[z.C](M, x.N, y.P) : [M/z]C} \vee\text{-E}$$

In the case(...) construct, the part $[z.C]$ is called the *motive*, because it is the motivation for performing the case analysis in the first place. What is different here from IPL is that C is parametric in $z : A + B$. The reason that this is necessary is that we need to be able to substitute the actual case we are analyzing (either inl or inr) in each branch into the type, so that our proof can vary. The case analysis proves C specialized to M, which is the term M flowing into the *type* C. The parametricity in z allows this dependency.

We can specialize this rule to the booleans, to obtain a new construct if:

$$\frac{\Gamma \vdash N : [\mathsf{tt}\,/z]C \quad \Gamma \vdash P : [\mathsf{ff}\,/z]C \quad \Gamma \vdash M : 2 \quad \Gamma, z : 2 \vdash C\ \mathsf{type}}{\Gamma \vdash \mathsf{if}[z.C](M, N, P) : [M/z]C}\ \vee\text{-E}$$

where the terms N and P do not bind any variables because by (η), the variables in the $\vee$-E rule are both just the null tuple, and thus don't actually need to be bound.

This presentation seems justified in the logical interpretation, but in programming it can lead to some unexpected results. In most programming languages, $\mathsf{if}(x, 17, tt)$ would not be well typed, as the true branch has type Nat and the false branch has type 2. With the tools of dependent type theory, however, we can give this term a type. If we posit the existence of conditionals in the type level, which we have not formalized yet, we can imagine the typing:

$$\mathsf{if}(M, 17, \mathsf{tt}) : \mathsf{if}(M, \mathsf{Nat}, 2)$$

for some suitable if construct at the type level, and noting that the "2" is a type name, not $s(s(z))$. Even though the two branches have different simple types, $\mathsf{if}(M, \mathsf{Nat}, 2)$ represents a join point for the two expressions. In normal programming languages, the join point type is not allowed to depend on any term, let alone the actual branch that was taken. This example then seems ill-typed, but in fact $17 : \mathsf{if}(\mathsf{tt}, \mathsf{Nat}, 2)$ and $\mathsf{tt} : \mathsf{if}(\mathsf{ff}, \mathsf{Nat}, 2)$, which are the critical premises in the $\vee$-E rule. This kind of construct is important when encoding $A + B$ using sigma types.

Finally, we can take some β and η rules as well, which mirror the non-dependent case. Here we omit the motive as it plays no role in the rules:

$$\begin{aligned}
&\mathsf{case}(\mathsf{inl}(M), x.N, y.P) \equiv [M/x]N && \beta_1 \\
&\mathsf{case}(\mathsf{inr}(M), x.N, y.P) \equiv [M/y]P && \beta_2 \\
&\mathsf{case}(M, x.[\mathsf{inl}(x)/z]P, y.[\mathsf{inr}(y)/z]P) \equiv [M/z]P && \eta
\end{aligned}$$

1.3.2 Natural Numbers

Like sum types, there is a relatively straightforward generalization of the elimination rule for naturals in terms of a recursor augmented with a motive:

$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma, z : \mathsf{Nat} \vdash C\ \mathsf{type} \quad \Gamma \vdash N : [0/z]C \quad \Gamma, x : \mathsf{Nat}, y : [x/z]C \vdash P : [s(x)/z]C}{\Gamma \vdash \mathsf{rec}[z.C](M, N, x.y.P) : [M/z]C}\ \mathsf{Nat\text{-}E}$$

Just as in the sum case, we can interpret the recursor as a proof of C for the specific natural M, given N which is a proof for zero and P which proves C for the successor

of some natural. In Gödel's T, we could show that omitting the variable $x : Nat$ was possible because in the presence of products we could add it if necessary. In this formulation, however, the type of y is a proof of C for some natural, so we need to give that natural a name to form the type.

We also have the (β) and (η) rules as expected:

$$\mathsf{rec}[z.C](0, N, x.y.P) \equiv N \quad \beta_1$$

$$\mathsf{rec}[z.C](s(M), N, x.y.P) \equiv [M, \mathsf{rec}[z.C](M, N, x.y.P)/x, y]P \quad \beta_2$$

$$\frac{[0/w]M \equiv N \quad [s(w)/w]M \equiv [w, M/x, y]P}{M \equiv \mathsf{rec}[z.C](w, N, x.y.P)} \quad \eta$$

1.3.3 Sigma elimination

We can characterize the sigma type elimination without fst and snd using a "degenerate form" of pattern matching where we have only one case:

$$\frac{\Gamma \vdash M : \Sigma x : A.B \quad \Gamma, z : \Sigma x : A.B \vdash C\ \mathsf{type} \quad \Gamma, x : A, y : B \vdash P : [\langle x, y\rangle / z]C}{\Gamma \vdash \mathsf{split}[z.C](M, x.y.P) : [M/z]C}\ \Sigma\text{-E}$$

We can take the following beta rule:

$$\mathsf{split}[z.C](\langle M_1, M_2\rangle, x.y.P) \equiv [M_1, M_2/x, y]P \quad (\beta)$$

We leave as an exercise showing that one can implement split from fst and snd. We can also show the converse, that fst and snd are definable in terms of split.

2 Identity Types

With this background, we can begin to talk meaningfully about the identity type. We say that the type $\mathsf{Id}_A(M, N)$ is the identity type of M and N in A, where $A\,\mathsf{type}$, $M : A$, and $N : A$. We can think of terms of this type as *proofs* that M and N are equal as elements of A. Crucially, we need dependence to even form this type. The formation rule is:

$$\frac{\Gamma \vdash A\,\mathsf{type} \quad \Gamma \vdash M : A \quad \Gamma \vdash N : A}{\Gamma \vdash \mathsf{Id}_A(M, N)\,\mathsf{type}}\ \mathsf{Id}\text{-F}$$

This type may also be written $M =_A N$. Confusingly, it may also be referred to as "propositional equality," to distinguish it from definitional equality. The simplest

element of this type is a proof of reflexivity, which has the introduction rule:

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

The elimination rule is:

$$\frac{\Gamma \vdash P : \mathsf{Id}_A(M, N) \quad \Gamma, x{:}A, y{:}A, z{:}\mathsf{Id}_A(x, y) \vdash C\ \mathsf{type} \quad \Gamma, x{:}A \vdash Q : [x, x, \mathsf{refl}_A(x)/x, y, z]C}{\Gamma \vdash \mathsf{J}[x.y.z.C](P, x.Q) : [M, N, P/x, y, z]C}\ \mathsf{Id\text{-}E}$$

One way to think of the identity type for a particular A is as an inductively generated family of types, where the induction is taken simultaneously over the two terms and the proof that they are equal (corresponding to x, y, and z in the rule). This view leads us to call the elimination rule *path induction*, where elements of type Id are "paths." Then M and N are the *endpoints* of this path. We call this an induction because we can prove C for any two endpoints and proof of their equality if we can prove C generically for some endpoint using Q. We need to consider a generic x in Q because the whole family Id_A is inductively generated at once, so we can't fix the endpoints M and N while proving the "inductive step." Further, because we only have one introduction rule, we can reason based on the fact that every path is just reflexivity, so Q can make this assumption.

J also admits a β-like rule:

$$\mathsf{J}[x.y.z.C](\mathsf{refl}_A(M); x.Q) \equiv [M/x]Q : [M, M, \mathsf{refl}_A(M)/x, y, z]C \quad (\beta)$$

2.1 Identity Types as Equivalence Relations

If Id is supposed to be equality, then we would like it to satisfy the three conditions of an equivalence relation: reflexivity, symmetry, and transitivity. Reflexivity is already part of the definition, but the other two properties are not part of the definition. We would like *proofs* that identity types satisfy these properties. Because we are working in a proof-relevant system, these proofs can also be viewed as enriched functional programs. Symmetry states that $x = y \implies y = x$, so a proof of symmetry is a simply map from $\mathsf{Id}_A(x, y)$ to $\mathsf{Id}_A(y, x)$. We can write the type of such a map formally as $\Pi x, y{:}A.\, \mathsf{Id}_A(x, y) \to \mathsf{Id}_A(y, x)$ (which is shorthand for $\Pi x, y{:}A.\, \Pi p{:}\mathsf{Id}_A(x, y).\, \mathsf{Id}_A(y, x)$). A proof of symmetry is just a *program* which has this type:

$$\mathrm{sym}_A := \lambda x{:}A.\lambda y{:}A.\lambda p{:}\mathsf{Id}_A(x, y).\mathsf{J}[x.y.z.\mathsf{Id}_A(y, x)](p; x.\mathsf{refl}_A(x))$$

The construction of symmetry is a function (using three λ's) whose body just immediately invokes path induction. The motive just states the overall conclusion, and

the proof Q is just the only way we have to generate elements of Id_A. The reason this proof is simple is any path is just reflexivity. The relation defined by the identity type is effectively the diagonal relation which only relates things to themselves, about which it is easy to prove symmetry.

Transitivity is more involved. Here we have not one, but two equality proofs, but J will only let us inspect one at a time. Like symmetry, we write trans_A as a map from elements and proofs of their equality to another proof of equality:

$$\begin{aligned}\mathrm{trans}_A :=& \lambda x, y, z{:}A.\ \lambda u : \mathsf{Id}_A(x, y).\ \lambda v{:}\mathsf{Id}_A(y, z).\\ &(\mathsf{J}[m.n.u.\ \mathsf{Id}_A(n, z) \to \mathsf{Id}_A(m, z)](u; m.\lambda v.v))(v)\end{aligned}$$

The way to understand this is that we need to inductively define a function, which will show that if y is equal to z, then x is equal to z. Then we can apply this function to the proof v, and obtain the desired result. For Q, we see that the motive degenerates to $\mathsf{Id}_A(m, z) \to \mathsf{Id}_A(m, z)$, so we can just write the identity function.

These specific proofs of symmetry and transitivity induce a kind of β-like behavior for definitional equality. In particular, due to the β rule for J, we see that

$$\mathrm{sym}_A(M)(M)(\mathsf{refl}_A(M)) \equiv \mathsf{refl}_A(M)$$

Further,

$$\mathrm{trans}_A(X)(X)(Z)(\mathsf{refl}_A(X))(Q) \equiv Q$$

because the induction J is defined on u, and thus when u is a reflexivity the whole J term reduces to the identity function. This definitional equivalence depends on the *how* we proved trans_A. For example, a different proof might not make the same equivalences, which can introduce dependencies between a proof and the exact definition of its lemmas. From a programming perspective, this is anti-modular.

2.2 Simple Functionality

In addition being an equivalence relation, we might hope that maps respect equality. Suppose we have a map F from A to B (so $x{:}A \vdash F : B$). In the non-dependent case, B will not depend on x, so we can can ask for some term to satisfy:

$$x, y{:}A, u{:}\mathsf{Id}_A(x, y) \vdash \_ : \mathsf{Id}_B(Fx, Fy)$$

We introduce the term $\mathsf{ap}Fu$ to be the lifting of F from a map between terms to a map between paths. ap may also be known as the "functorial action." We can define ap using a path induction as:

$$\mathsf{ap}\ F\ u := \mathsf{J}[x.y.\_.\mathsf{Id}_B(Fx, Fy)](u; x.\mathsf{refl}_B(Fx))$$

2.3 Transportation

If we have a dependent map, then the above is no longer sufficient. In this case, we have that $z{:}A \vdash B\ \mathsf{type}$ and $x{:}A \vdash F : B$ as before. Crucially different is that B may depend on x, so Fx and Fy won't even necessarily have the same type! Thus we can't even form $\mathsf{Id}_B(Fx, Fy)$. If $x :\equiv y$, then we would know that the two destination types are the same by substitution, but we only have that x is equal to y *propositionally*. We would expect that $B[x]$ and $B[y]$ to be *related* in some way, as but we do not go as far as to say the two spaces are the same.

What we will do is say that if two elements x and y have a path p in A between x and y (i.e. $p : \mathsf{Id}_A(x, y)$), then there is a lifting of this path, p_*, to act as a transport map between $[x/z]B$ and $[y/z]B$. Here A is the "base space," and all B taken over every element of A is known as the "total space." B for some z is known as a fiber.

We introduce the transport tr, with the following specification:

$$x, y{:}A,\ p{:}\mathsf{Id}_A(x, y) \vdash \mathsf{tr}[z.B](p) : [x/z]B \to [y/z]B$$

Informally, this says that a transport, which is specified by a total space ($z.B$) and a path p, takes elements from the fiber associated with the start of the path to the end-of-path fiber. We can define transport to be:

$$\mathsf{tr}[z.B](p) := \mathsf{J}[m.n._.[m/z]B \to [n/z]B](p; m.\lambda w.w)$$

Because the motive is $[m/z]B \to [n/z]B$, we can use the identity function inside J as them motive will become $[m/z]B \to [m/z]B$. Outside, however, because $p : \mathsf{Id}_A(x, y)$, this will become $[x/z]B \to [y/z]B$. The intuition here is that because the two elements x and y are "equal," we don't need anything more than a glorified identity function to transport between the induced fibers.

15-819 Homotopy Type Theory Lecture Notes

Nathan Fulton

October 9 and 11, 2013

1 Contents

These notes summarize and extend two lectures from Bob Harper's Homotopy Type Theory course. The cumulative hierarchy of type universes, Extensional Type theory, the ∞-groupoid structure of types and iterated identity types are presented.

2 Motivation and Overview

Recall from previous lectures the definitions of functionality and transport. Functionality states that functions preserve identity; that is, domain elements equal in their type map to equal elements in the codomain. Transportation states the same for type families. Traditionally, this means that if $a =_A a'$, then $B[a]$ true iff $B[a']$ true. In proof-relevant mathematics, this logical equivalence is generalized to a statement about identity in the family: if $a =_A a'$, then $B[a] =_B B[a']$.

Transportation can be thought of in terms of functional extensionality. Unfortunately, extensionality fails in ITT. One way to recover extensionality, which comports with traditional mathematics, is to reduce all identity to reflexivity. This approach, called Extensional Type theory (ETT), provides a natural setting for set-level mathematics.

The HoTT perspective on ETT is that the path structure of types need not be limited to that of strict sets. The richer path structure of an ∞-groupoid is induced by the induction principle for identity types. Finding a type-theoretic description of this behavior (that is, introduction, elimination and computation rules which comport with Gentzen's Inversion Principle) is an open problem.

3 The Cumulative Hierarchy of Universes

In previous formulations of ITT, we used the judgement $A\,\mathsf{type}$ when forming types. In this setting, many types are natural to write down but impossible to form. As a running example for the section, consider the following: $\mathsf{if}\, M, \overline{17}, \mathsf{tt} : \mathsf{if}\, M, \mathsf{Nat}, \mathsf{Bool}$. Assuming the well-formedness of the type, elimination rules behave as expected: $\overline{17} : \mathsf{if}\,\mathsf{tt}, \mathsf{Nat}, \mathsf{Bool} \equiv \mathsf{Nat}$ and $\mathsf{tt} : \mathsf{if}\,\mathsf{ff}, \mathsf{Nat}, \mathsf{Bool} \equiv \mathsf{Bool}$.

Forming this type is not possible using the current formation rule for $\overline{\mathsf{if}}$. Type universes address this shortcoming by generalizing type formation rules. A recursively generated cumulative hierarchy of universes ($\mathcal{U}_i$) is introduced. Instead of defining type formation in terms of a judgement $A\,\mathsf{type}$, formation rules state the relative location of relevant types in the hierarchy; that is, judgements that $A\,\mathsf{type}$ are replaced with judgements of the form $A : \mathcal{U}_i$.

The definition of type universes includes three new rules[1]

$$\frac{\Gamma\,\mathsf{ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-intro} \qquad \frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-cumul} \qquad \frac{A \equiv B : \mathcal{U}_i}{A \equiv B : \mathcal{U}_{i+1}}\ \mathcal{U}\text{-}\equiv$$

The $\mathcal{U}$-intro rule introduces an unbounded hierarchy of universes, each of which inhabits the next universe. The second rule states that these universes are cumulative, and the third ensures that equality is preserved in higher universes. The $\mathcal{U}$-$\equiv$ rule is a derived rule in the HoTT book presentation.

In addition to these rules, every type formation rule establishes relative positions of relevant types. For example[2]:

$$\frac{A : \mathcal{U}_i \quad M, N : A}{Id_A M, N : \mathcal{U}_i}\ \mathcal{U}\text{Id-F} \qquad \frac{A : \mathcal{U}_i \quad x : A \vdash B : \mathcal{U}_i}{\Pi_{x:A} B : \mathcal{U}_i}\ \mathcal{U}\Pi\text{-F}$$

$$\frac{\Gamma\,\mathsf{ctx}}{1 : \mathcal{U}_i}\ \mathcal{U}1\text{-F} \qquad \frac{\Gamma\,\mathsf{ctx}}{0 : \mathcal{U}}\ \mathcal{U}0\text{-F} \qquad \frac{A : \mathcal{U} \quad B : \mathcal{U}}{A + B : \mathcal{U}}\ \mathcal{U} + \text{-F}$$

The addition of universes to ITT solves the problem identified by the running example. As the example suggests, these hierarchies increase the expressiveness of ITT. This is established by identifying a statement that cannot be proven only in the presence of universes.

[1] See A1.1 and A2.3 of the HOTT book for discussion.

[2] See appendix 2 of [2] for a full formulation

example: Jan Smith established that without universes, it is not provable that $\mathsf{succ}(-) \neq 0$ [5]. However, in Martin-Löf's Type Theory, it is provable that $n : \mathsf{Nat}$, $\mathsf{succ}(n) \not\equiv 0$[3]. In fact, Smith proved that *any* negation of an equivalence cannot be proven without universes.

exercise: Show that the operators fst and snd can be defined from split.

3.1 Typical Ambiguity

In the examples above, subscripts on each universe create significant notational overhead. Therefore, these indices are elided whenever intent is obvious. When implemented with pen and paper, this is called **typical ambiguity**. Its mechanization in Coq is referred to as **universe polymorphism**.

3.2 Alternatives to the Hierarchy

The introduction of an infinite hierarchy of universes complicates the theory. An uninitiated reader might wonder whether an infinite, cumulative hierarchy is really necessary. This section presents three alternatives. The first alternative works, but has some disadvantages. The other two alternatives have significant problems, demonstrating that the complexity induced by type universes is essential to a consistent and sufficiently expressive definition of ITT.

3.2.1 Large Elimination

Intensional Type Theory can be consistently formulated without a hierarchy. The approach, called Large Elimination, rules the correct types into the theory by hand. In the case of the running example, the rule would be:

$$\frac{M : Bool \quad A\ \mathsf{type} \quad B\ \mathsf{type}}{\mathsf{if}\, M, A, B\ \mathsf{type}}\ \text{LE-If}$$

Similar rules must be provided for each type formation rule. The universal approach is preferred because it is less ad hoc —large elimination requires the addition of new rules for each affected type.

[3] See page 86 of Programming in Martin-Löf's Type Theory [1].

3.2.2 A Single Universe

The running example may be addressed without introducing a recursively defined hierarchy of universes. One alternative is to replace the $\mathcal{U}$ rules above with a single universe. In this case, the important choice is whether $\mathcal{U} : \mathcal{U}$.

If the universe is not self-inclusive, The formulation problem discussed above re-emerges. For instance, the type $\mathsf{if}\, M, \mathcal{U}, \mathcal{U} \to \mathcal{U}$ is not formable without a recursively defined hierarchy. The same observation applies at the top of any finite hierarchy.

3.2.3 The Inconsistent Approach

An insightful reader might observe that this problem can be resolved by patching the single universe system with a rule which allows the universe to contain itself:

$$\frac{}{\Gamma \vdash \mathcal{U} : \mathcal{U}}\ \mathcal{U}\text{-inconsistent}$$

This system allows the formation of $\mathsf{if}\, M, \mathcal{U}, \mathcal{U} \to \mathcal{U}$. However, it also destroys the consistency of the theory.

Exercise: Reproduce the Burali-Forte Paradox within a system equipped with U-cumul-inconsistent[4].

4 Proof-relevance and Extensionality

Martin-Löf's Type Theory is significant because it introduces the notion of *proof relevance*. Intuitively, this expresses the idea that proofs can be treated as mathematical objects.

4.1 The Theorem of Choice

It is well-known that the Axiom of Choice is independent of the axioms of set theory. However, choice can be derived in ITT. The derivation provides an excellent example of proof relevance in action.

The theorem of choice states that if xCy is total, then there must exist a function (f) which associates each x with a chosen $y = f(x)$. We can state this formally in ITT.

Theorem of Choice. $\vdash e : \Pi_{x:A}\Sigma_{y:B}C(x,y) \to \Sigma_{f:A\to B}.\Pi_{x:A}C(x,f(x))$.

[4]This formulation of the paradox is due to Girard 1972, and is referred to as Girard's Paradox.

The proof, provided in [4], involves finding a derivation of:

$$F : \Pi_{x:A}\Sigma_{y:B}C(x,y) \vdash \lambda F. < \lambda x.\mathsf{fst}F(x), \lambda x.\mathsf{snd}F(x) >: \Sigma_{f:A\to B}.\Pi_{x:A}C(x,f(x))$$

In the proof, F is both an assumption and a mathematical object (namely, a product). Therefore, the proof may rely upon not only the inhabitation of the type of F, but also F itself.

Note: In future lectures, banana brackets will be used to recover a more traditional reading of F by suppressing the ability to use it as a piece of data in the proof. For now, the significant observation is that proof irrelevance can be recovered within ITT.

4.2 Failure of Extensionality

Marin-Löf demonstrated that the Axiom of Extensionality fails in ITT; in ITT, it is not the case that if $p : Id_A(M,N)$ for closed M, N, A, then $M \equiv N : A$.

Extensional Type theory (ETT) addresses the failure of extensionality in ITT by endowing the theory with the principle of equality of reflection. Concretely, ETT introduces two new rules which reduce identity to equivalence. Therefore, all identity paths on a type are the reflexive path.

$$\frac{\Gamma \vdash p : Id_A(M,N)}{\Gamma \vdash M \equiv N : A}\ \text{Eq-Refl} \qquad \frac{p : Id_A(M,N)}{\Gamma \vdash p \equiv \mathsf{refl}(M) : Id_A(M,M)}\ \text{UIP}$$

The first rule, equality of reflection, states that proof of an identification is sufficient to show judgemental equality in the type. The second rule, Uniqueness of Identity Proofs, states that any path is the reflexive path.

Although extensionality does not hold generally for ITT, uniqueness of identity proofs may be recovered for a large class of types.

Hedberg's Theorem. *Any set with decidable identities has collapsed identity sets [3][5].*

4.2.1 ETT vs ITT

The essential difference between ETT and ITT is the algebraic structure of types. ETT reduces all identity paths to reflexivity. As a result, the path structure of types

[5]There is a Coq proof by Nicolai Kraus online: http://www.cs.nott.ac.uk/~ngk/hedberg_direct.v

in ETT is homotopically discrete. ITT admits a much richer path structure on types: two paths $p : Id_A(A, B)$ and $q : Id_A(B, C)$ may by equal but not trivially equal.

The two other major differences between ETT and ITT are decidability of type checking and fitness for set-level mathematics.

The UIP rules introduces proof search as a valid mode of operation for the type checker. Therefore, type checking is not decidable in ETT[6]. Decidability is not an important criterion for two reasons. First, the standard mode of operation in a mechanized ETT (e.g. NuPRL) does not result in proof search. Second, type checking in ITT quickly becomes intractable.

A more important secondary distinction between ETT and ITT is fitness for set- level mathematics. Types in ETT have the structure of an h-set; therefore, set-level mathematics is much nicer in NuPRL than in Coq. Whereas extensionality and transport come for free in NuPRL, Coq users must induce this structure by programming in terms of a setoid. However, the convienance of ETT comes at a cost: the path structure of its types in necessarily limited due to Hedberg's Theroem.

Just as proof-relevant mathematics subsumes proof-irrelevant mathematics as a special case, the ∞-groupoid structure of types in ITT may be forgotten so that ETT is recovered as a special case. In fact, this is essentially what happens with Setoid in Coq.

5 Algebraic Structure of Identity Types

Recall that the induction principle for identity types states that for $x, y : A$, there exists an identity type $x =_A y$. Furthermore, proving a property for these elements and a path $p : x =_A y$ consists of proving the property in the reflexive case (that is, for x, x, refl_x).

The full implications of this principle were not understood when it was first introduced. A realization central to Homotopy Type Theory is that the induction principle for identity types gives rise to an entire hierarchy of iterated identity types. That is, due J, we can form the type $p =_{\mathsf{Id}_A(x,y)} q$ and so on. Homotopy Type Theory is so-called, in part, because these types form the same structure as iterated homotopies: that of an $\infty-$groupoid.

Whereas the universes provide a mechanism for reasoning about size in an iterative fashion, the iterated identities provide an account of dimension. In the example above, x and y are start and end points. The first identity corresponds to a path between

[6]Type checking for ITT is decidable

the elements. Paths between $p, q : \mathsf{Id}_{\mathsf{Id}_A(x,y)}$ are homotopies, or 2-dimensional paths. Each iteration corresponds to an increase in dimension[7].

Before proceeding with a presentation of the groupoid axioms in terms of ITT, it is useful to recall the derivable equivalence relation:

(1) $id_A(M) := \mathsf{refl}_A(M) : Id_A(M, M)$
(2) $p : Id_A(M, N) \vdash p^{-1} : Id_A(N, M)$
(3) $p : Id_A(M, N), q : id_A(N, P) \vdash p \centerdot q : Id_A(M, P)$

The second and third are theorems provable by path induction, since composition and inversion are both defined in terms of J. Therefore, we may read identity types propositionally as witnesses of an equivalence, and computationally as abstract daata types upon which we may operate. The two notations for identity types, $x =_A y$ and $\mathsf{Id}_{A(x,y)}$, typically elucidate the intended reading. While the former reading comports with more traditional interpretations of equality, the latter gives rise to the iterated identity types.

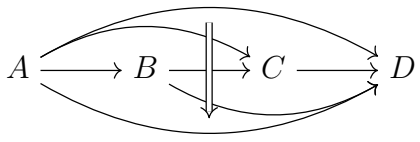
Remark. *Despite the correspondence with classical (analytic) homotopy theory, maps should be thought of synthetically. As a result, path concatenation is not defined in terms of function composition; hence, the* $\centerdot$ *notation.*

5.1 The Groupoid Laws

The groupoid laws may be forumated as coherence theorems, each proven by path induction using J:

inv-right	$p \centerdot p_{-1} =_{Id_A(M,M)} \mathsf{id}(M)$
inv-left	$p^{-1} \centerdot p =_{Id_A(N,N)} \mathsf{id}(N)$
unit-right	$p \centerdot id(N) =_{Id_A(M,N)} p$
unit-left	$id(M) \centerdot p =_{Id_A(M,N)} p$
assoc	$(p \centerdot q) \centerdot r =_{Id_A(M,P)} p \centerdot (q \centerdot r)$

Before proceeding an explanation of how these are proven, some motivation may be helpful. Consider the following diagram for the associativity theorem:



This diagram illuminates the weak nature of the path structure: associativity holds only because it holds at yet higher type. Iterated identity types are given structure by this higher coherence.

[7]Dimension is also referred to as homotopy level.

Theorem 1. *The groupoid laws hold.*

Full proofs are available in chapter 2 of [2]. We outline portions of the argument here for the sake of later discussion. For the first inverse theorem, perform path induction on p. It suffices to consider that $p = \mathsf{refl}_x$. We have by the definition of refl_x^{-1} that $\mathsf{refl}_x \centerdot \mathsf{refl}_x^{-1} = \mathsf{refl}_x$. The other inverse argument follows similarly. The cases for the unit theorems and associativity are similar. Each follows by path induction on p, considering the case where $p = \mathsf{refl}_x$.

5.2 Maps preserve structure

Given this structure, it is natural to ask whether mappings preserve the groupoid structure. Recall that ap preserves identities.

Theorem 2. *If* $f : A \to B$ *and* $p : M =_A M'$ *then* $\mathsf{ap}_f(p) : fM =_B fM'$.

Mappings preserve not just identity, but the entire groupoid structure. Proving this requires showing that *ap* preserves identity, inversion and composition. That is,

Theorem 3. *For a function* $f : A \to B$ *and paths* $p : x =_A y$, $q : y =_A z$

1) $\mathsf{ap}_f(\mathsf{refl}(x)) \equiv \mathsf{refl}(f(x))$

2) $\mathsf{ap}_f(p^{-1}) = ap_f(p)^{-1}$

3) $\mathsf{ap}_f(p \centerdot q) =_{\mathsf{Id}_a(x,z)} ap_f(p) \centerdot ap_f(q)$

Exercse: Prove that maps preserve functoriality. A formal proof will be included in a coming revision of these notes.

5.3 Does Homotopy Type Theory have a Computational Interpretation?

In the sketch of the groupoid proofs, the general case of p is reduced to the case of refl_x. Currently, this is justified by a categorical model. This is not natural or desirable because the distinguishing characteristic of type theory is its computational content characterized by Gentzen's Inversion Principle. The model-based justification is insufficient, in part, because it does not provide a way of *running* HoTT programs. The constructivity of Homotopy Type Theory is important because Hedberg's Theorem collapses the dimensional tower developed in this section. However, determining whether there is a computational interpretation of Homotopy Type theory is a principle open problem.

References

[1] Jan M. Smith Bengt Nodström, Kent Petersson. Programming in martin-löf's type theory. `http://www.cse.chalmers.se/research/group/logic/book/book.pdf`, 1990.

[2] Institute for Advanced Study. *Homotopy Type Theory: Univalent Foundations of Mathematics*. The Univalent Foundations Program, 2013. `http://homotopytypetheory.org/book/`.

[3] Michael Hedberg. A coherence theorem for martin-löf's type theory. *J. Funct. Program.*, 8(4):413–436, July 1998.

[4] Per Martin-Löf. Intuitionistic type theory. `http://intuitionistic.files.wordpress.com/2010/07/martin-lof-tt.pdf`, 1980.

[5] Jan M. Smith. An interpretation of martin-lof's type theory in a type-free theory of propositions. *J. Symb. Log.*, 49(3):730–753, 1984.

15-819 Homotopy Type Theory Lecture Notes

Robert Lewis and Joseph Tassarotti

October 21 and 23, 2013

1 Paths-over-Paths

Recall that last time we explored the higher groupoid structure of types, and showed that for non-dependent maps, ap preserves this structure. Now, in the case where we have a dependent function $f : \Pi x : A.B$, we would like to similarly state that f maps equals to equals, so that given a path $p : \mathsf{Id}_A(M, N)$, there is some map which takes in p and gives a path between fM and fN. However, because f is dependent, $fM : [M/x]B$ and $fN : [N/x]B$. Although these types are related, they are not equal, so we cannot talk about propositional equality between fM and fN.

In earlier lectures, we defined $\mathsf{tr}[x.B]p : [M/x]B \to [N/x]B$, often written as p_*, which lifts the path p to a mapping between the fibers $[M/x]B$ and $[N/x]B$. Since $p_*(fM)$ and fN share the same type, we can meaningfully talk about equality between them. We can now define a map $\mathsf{apd}_f : \Pi p : \mathsf{Id}_A(M, N).\mathsf{Id}_{[N/x]B}(p_*(fM), fN)$ by

$$\mathsf{apd}_f p := \mathsf{J}[m.n.z.\mathsf{Id}_{[n/x]B}(z_*(fm), fn)](p; m.\mathsf{refl}_{[m/x]B}(m))$$

This has the appropriate type because when the path is simply $\mathsf{refl}_A(M)$, we have that $(\mathsf{refl}_A(M))_* \equiv \mathsf{refl}_{[M/x]B}(fM)$. See figure 1 for a pictorial representation of this.

Now, since p_*^{-1} gives a map between the fibers going the other way, we could just as well have defined an analogous term $\mathsf{apd}'_f : \mathsf{Id}_A(M, N) \to \mathsf{Id}_{[M/x]B}(fM, p_*^{-1}(fN))$. Moreover, we have that

$$\begin{aligned} \mathsf{ap}_{p_*^{-1}}(\mathsf{apd}_f p) \quad &: \quad \mathsf{Id}_{[M/x]B}(p_*^{-1}(p_*(fM)), p_*^{-1}(fN)) \\ &\equiv \mathsf{Id}_{[M/x]B}(fM, p_*^{-1}(fN)) \end{aligned}$$

$$\begin{aligned} \mathsf{ap}_{p_*}(\mathsf{apd}_f p^{-1}) \quad &: \quad \mathsf{Id}_{[N/x]B}(p_*(fM)), p_*(p_*^{-1}(fN))) \\ &\equiv \mathsf{Id}_{[N/x]B}(p_*(fM), fN) \end{aligned}$$

which shows that these two theorems imply one another.

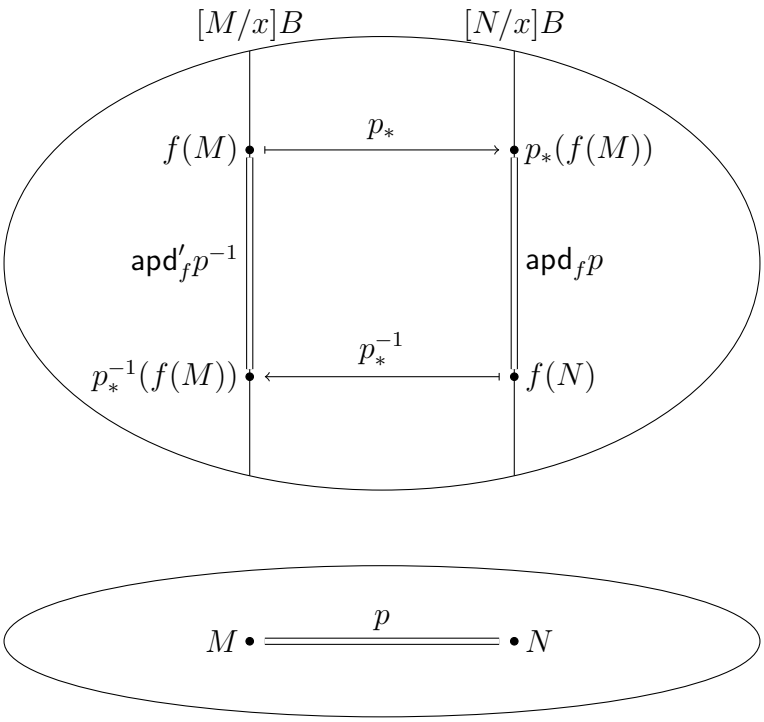


Figure 1: Paths-over-paths

The lack of symmetry in the types of apd_f and apd'_f is somewhat awkward when developing machine checked proofs. It's more convenient to define a symmetric notation, $f(M) =_p^{x.B} f(N) :\equiv Id[N/x]B(p_*(fM), fN)$, which we read as "$f(M)$ and $f(N)$ are correlated by p" . This corresponds to the type of paths *over* the path p. Using this notation, we can prove theorems about this type like:

$$\mathrm{sym}_{\mathrm{corr}} \quad : \quad Q =_p^{x.B} R \;\to\; R =_{p^{-1}}^{x.B} Q$$

$$\mathrm{trans}_{\mathrm{corr}} \quad : \quad Q =_p^{x.B} R \;\to R =_q^{x.B} S \to Q =_{p \cdot q}^{x.B} S$$

2 Equivalence of Types

2.1 Motivation

We start by informally recalling some notions of equivalence that are commonly used in mathematics:

1. *Biconditional propositions*: Given two propositions p and q such that $p \supset q$ and $q \supset p$, we might wish to say that $p = q$, because these two propositions are logically equivalent. In classical logic, this makes sense, because p and q are both either equal to true or equal false. To quote Whitehead and Russell [3, p.115]:

   > When each of two propositions implies the other, we say that the two are *equivalent*, which we write "$p \equiv q$" ... It is obvious that two propositions are equivalent when, and only when, both are true or both are false...
   >
   > We shall give the name of a *truth-function* to a function $f(p)$ whose argument is a proposition, and whose truth-value depends only upon the truth-value of its argument. All the functions of a proposition with which we shall be specially concerned will be truth-functions, *i.e.* we shall have
   >
   > $$p \equiv q \,.\supset\, .f(p) \equiv f(q).$$

   This means that for Whitehead and Russell, if p and q are logically equivalent, then they are indiscernible. However, in the proof relevant setting of type theory, this is not the case, because these types classify particular pieces of data. Although terms of the type $f : p \to q$ and $g : q \to p$ give us ways to interconvert proofs of p and q, a proof of p is not by itself a proof of q. Moreover, it need not even be the case that f and g are inverses of each other.
2. *Isomorphic sets*: In set theory, we say that two sets A and B are *isomorphic* if there is a bijection between them. That is, there are functions $f : A \to B$ and $g : B \to A$ such that $g(f(a)) = a$ and $f(g(b)) = b$. In many contexts, it is not relevant for us to distinguish between isomorphic sets. However, in ZF set theory, just because two sets are isomorphic does not mean they are indiscernible, so we cannot regard them as equal.
   This is a larger symptom of the fact that although ZF set theory lets us encode the structures of mathematics, it does not support abstraction. Propositions like $0 \in 1$ are perfectly well-formed, and are even true for most encodings of the natural numbers as sets. As de Bruijn points out [1], these artifacts of a particular

encoding contradict the way we conceptually think of mathematics[1]:

> In our mathematical culture we have learned to keep things apart. If we have a rational number and a set of points in the Euclidean plane, we cannot even imagine what it means to form the intersection. The idea that both might have been coded in ZF with a coding so crazy that the intersection is *not empty* seems to be ridiculous...
>
> A very clear case of thinking in terms of types can be found in Hilbert's axiomatization of geometry. He started by saying that he assumes there are certain things which will be called *points* and certain things to be called *lines*. Nothing is said about the nature of these things.

Type theory rules out statements like $0 \in 1$ as ill-formed. As we shall see, this same facility for abstraction allows us to give a more suitable treatment of equivalence.

Now, we turn to the question of equivalences of types. Applying our naïve intuition of regarding types as sets, we might say that types are isomorphic precisely when there is a bijection between them. In ITT, this will work for types corresponding to first order data, but we encounter problems when considering functions.

More precisely, to show that $A \to B$ is isomorphic to $C \to D$, we need to construct functions $F : (A \to B) \to (C \to D)$ and $G : (C \to D) \to (A \to B)$ such that for all $f : A \to B$ and $g : C \to D$, $G(F(f)) = f$ and $F(G(g)) = g$. In a set-theoretic setting, it would suffice to show that for all $x \in A$, $G(F(f))(x) = f(x)$, and similarly for $F \circ G$. However, in ITT we lack function extensionality, this is not enough. We need to show that $G \circ F$ maps f precisely back to itself. One might try to resolve this by quotienting by extensionality or adding in an axiom of extensionality.

However, the problem becomes even more difficult when considering universes. We would need to show that for each type A in the universe, $G(F(A)) = A$. Just as with functions, where we were really interested in showing that $G \circ F$ mapped a function to something that was extensionally equivalent, here we want $G(F(A))$ to itself be isomorphic to A not equal.

2.2 Homotopy Equivalence

We now introduce the notion the notion of a *homotopy*. Given two functions $f, g : A \to B$, a homotopy from f to g is a term with type $\Pi x : A.\mathsf{Id}_B(fx, gx)$. We

[1] A portion of this passage is quoted in [2], which contains an interesting discussion about some advantages and disadvantages of types and sets.

introduce the notation $f \sim_{A\to B} g$ for the type of homotopies from f to g. If this type is inhabited, we say that f is "homotopic to" g.

Now, given $H : f \sim_{A\to B}$, we have that for all $x : A$, $fx =_B gx$. But in fact there is something more going on: H is *dependently functorial* in $x : A$. That is, H respects paths between inhabitants in A and B. This property is also called *naturality*; we can say that H is a sort of polymorphism in $x : A$. This means that the following diagram commutes:

$$\begin{array}{ccc} f(a) & \overset{H(a)}{=\!=} & g(a) \\ {\scriptstyle \mathsf{ap}_f(p)}\Big\| & & \Big\|{\scriptstyle \mathsf{ap}_g(p)} \\ f(a') & \underset{H(a')}{=\!=} & g(a') \end{array}$$

2.3 Basic Properties of Equivalence

For a function $f : A \to B$, we define an *equivalence* between A and B, by

$$\mathsf{isequiv}(f) :\equiv (\Sigma g : B \to A.f \circ g \sim \mathsf{id}_B) \times (\Sigma h : B \to A.h \circ f \sim \mathsf{id}_A).$$

The proposition expressing that two types A and B are equivalent, written $A \simeq B$ b, is

$$A \simeq B :\equiv \Sigma f : A \to B.\mathsf{isequiv}(f).$$

Since we are in a proof-relevant setting, the information that $A \simeq B$ consists of five things:

- A function $f : A \to B$
- A function $g : B \to A$
- A proof $\alpha : \Pi y : B.f(g(y)) =_B y$
- A function $h : B \to A$
- A proof $\beta : \Pi x : A.h(f(x)) =_A x$

To prove that $A \simeq B$, we need to provide all of these as evidence, and from evidence that $A \simeq B$, we can extract all of these.

We will also be interested in the notion of a *quasi-inverse*:

$$\mathsf{qinv}(f) :\equiv \Sigma g : B \to A.(f \circ g \sim \mathsf{id}_B \times g \circ f \sim \mathsf{id}_A).$$

As one might hope, the notions of equivalence and quasi-inverse are very closely related. One can prove the following properties:

1. For every $f : A \to B$, there is a function $\mathsf{qinv}(f) \to \mathsf{isequiv}(f)$.
2. For every $f : A \to B$, there is a function $\mathsf{isequiv}(f) \to \mathsf{qinv}(f)$.

This means that the two notions are logically equivalent: a function is an equivalence if and only if it has a quasi-inverse. In addition, we can show that $\mathsf{isequiv}(f)$ expresses an HPROP: that is, up to higher homotopy, there is only one proof of this fact. This will become important later.

2.4 Function extensionality

The axiom of function extensionality allows us to show that the $(f =_{A\to B} g) \simeq (f \sim_{A\to b} g)$. Even without the axiom, we can define the map

$$\mathsf{happly} : f =_{A\to B} g \to f \sim_{A\to B} g$$

Now, the axiom says that the above map is an equivalence: if we have a proof of $f \sim_{A\to B} g$, we may assume that we have a proof of $f =_{A\to B} g$. This is not necessarily provable without the axiom. For example, in the natural number type, $\lambda x.0 + x \sim_{N\to N} \lambda x.x$, but since addition was defined inductively on the second argument, we cannot find a path between them.

2.5 Exercises

The following propositions are left as exercises, with the first one begun for explanatory purposes:

1. Show that $\mathsf{id}_A : A \to A$ is an equivalence.
   To do this, we need four pieces of information:
   (a) $g : A \to A$. Take this to be id_A.
   (b) A proof $\alpha : \Pi y : A.\mathsf{id}_A(g(y)) =_A y$.
   (c) $h : A \to A$. Again, take this to be id_A.
   (d) A proof $\beta : \Pi x : A.h(\mathsf{id}_A(x)) =_A x$.
2. If $f : A \to B$ is an equivalence, then there is $f^{-1} : B \to A$ (given by the quasi-inverse of f) that is also an equivalence.
3. If $f : A \to B$ and $g : B \to C$ are equivalences, then so is $g \circ f : A \to C$.

3 Structure of Paths in Types

We want to examine the paths inside certain types. For the negative types, this will be relatively simple. For the positive types, it will be much harder. There are many outstanding open problems within the positive types.

3.1 Product Types

We start by examining the paths in $\mathsf{Id}_{A\times B}(\_,\_)$.

There is a function f such that

$$f : \mathsf{Id}_{A\times B}(x,y) \to (\mathsf{Id}_A(\pi_1 x, \pi_1 y) \times \mathsf{Id}_B(\pi_2 x, \pi_2 y)).$$

Specifically,

$$f :\equiv \lambda p.\langle \mathsf{ap}_{\pi_1}(p), \mathsf{ap}_{\pi_2}(p)\rangle$$

Roughly speaking, if $x =_{A\times B} y$, then $\pi_1 x =_A \pi_1 y$ and $\pi_2 x =_B \pi_2 y$.

Proposition. *f is an equivalence:* $\mathsf{Id}_{A\times B}(x,y) \simeq \mathsf{Id}_A(\pi_1 x, \pi_1 y) \times \mathsf{Id}_B(\pi_2 x, \pi_2 y)$.

Proof. As noted in Section 2, it suffices to produce a quasi-inverse for f. We need to construct three objects:

1. $g : (\mathsf{Id}_A(\pi_1 x, \pi_1 y) \times \mathsf{Id}_B(\pi_2 x, \pi_2 y)) \to \mathsf{Id}_{A\times B}(x,y)$
2. $\alpha : g(f(p)) =_{\mathsf{Id}_{A\times B}(x,y)} p$
3. $\beta : f(g(q)) =_{\mathsf{Id}_A(\pi_1 x,\pi_1 y)\times \mathsf{Id}_B(\pi_2 x,\pi_2 y)} q$

We construct these as follows:

1. We define two auxiliary functions

$$\mathsf{pair} :\equiv \lambda x \lambda y \langle x, y\rangle : A \to B \to A \times B$$

and

$$\mathsf{ap2}_f : \mathsf{Id}(x,x') \to \mathsf{Id}(y,y') \to \mathsf{Id}(fxy, fx'y')$$

Using these, we can then define

$$g :\equiv \lambda\langle p, q\rangle.\mathsf{ap2}_{\mathsf{pair}} pq$$

.

2. To define α, it suffices (by FUNEXT) to show:
   - $\eta : \Pi p(\mathsf{ap2}_{\mathsf{pair}}(\mathsf{ap}_{\pi_1}(p), \mathsf{ap}_{\pi_2}(p)) = p)$
   - $\beta_1 : \Pi p \Pi q(\mathsf{ap}_{\pi_1}(\mathsf{ap2}_{\mathsf{pair}} pq) = p)$
   - $\beta_2 : \Pi p \Pi q(\mathsf{ap}_{\pi_2}(\mathsf{ap2}_{\mathsf{pair}} pq) = q)$

   By path induction, we need to find R such that

$$x : A \times B \vdash R : (\mathsf{ap2}_{\mathsf{pair}}(\mathsf{ap}_{\pi_1}(\mathsf{refl}(x)), \mathsf{ap}_{\pi_2}(\mathsf{refl}(x)))) = \mathsf{refl}(x)$$

   Then,

$$\eta :\equiv J[\,](p; x.R).$$

By our earlier definition of ap^2, we have that

$$\begin{aligned}
\mathsf{ap}_{\pi_1}(\mathsf{refl}(x)) &\equiv \mathsf{refl}(\pi_1(x)) \\
\mathsf{ap}_{\pi_2}(\mathsf{refl}(x)) &\equiv \mathsf{refl}(\pi_2(x)), \text{ and from these,} \\
\mathsf{ap2}_{\mathsf{pair}}(\mathsf{refl}(\pi_1(x)))(\mathsf{refl}(\pi_2(x))) &\equiv \mathsf{refl}\langle \pi_1(x), \pi_2(x)\rangle \\
&\equiv \mathsf{refl}(x)
\end{aligned}$$

3. The constructions of β_1 and β_2 are similar and left as exercises.

□

3.2 Coproduct Types

Similarly, we can look into $\mathsf{Id}_{A+B}(x, y)$. Intuitively speaking, we would like to say that any path in the space $A + B$ is either a path in A or a path in B; we would never expect to have a path (equation) between an $\mathsf{inl}(a)$ and an $\mathsf{inl}(b)$.

We would like to prove the following facts:

$$\begin{aligned}
\mathsf{Id}_{A+B}(\mathsf{inl}(a), \mathsf{inl}(a')) &\simeq \mathsf{Id}_A(a, a') \\
\mathsf{Id}_{A+B}(\mathsf{inr}(b), \mathsf{inr}(b')) &\simeq \mathsf{Id}_B(b, b') \\
\mathsf{Id}_{A+B}(\mathsf{inl}(a), \mathsf{inr}(b)) &\simeq 0 \\
\mathsf{Id}_{A+B}(\mathsf{inr}(a), \mathsf{inl}(b)) &\simeq 0
\end{aligned}$$

Proving this requires a bit of a trick.

Suppose we wanted to prove the first equivalence alone. The right-to-left direction is simple. For the left-to-right direction, we need to exhibit

$$p : \mathsf{Id}_{A+B}(\mathsf{inl}(a), inl(a')) \vdash R : \mathsf{Id}_A(a, a').$$

R must be a path induction on p, of the form $R = J[C](p, \_)$ for some motive C. The conclusion of this path induction will be of the form $C(\mathsf{inl}(a), \mathsf{inl}(a'), p)$. But what we need is $\mathsf{Id}_A(a, a')$ (note the lack of inl). One might try to define something like $D(u, v) = \mathsf{Id}_A(\mathsf{outl}(u), \mathsf{outl}(v))$, but this cannot exist, since outl cannot be a total function.

[2] This is a striking example of anti-modularity. One has no reason to expect that this equality should hold definitionally; it depends essentially on how ap was defined, not just on its type. It would be nice to avoid this kind of code-on-code dependency, since "the proof should not have to know about the computation."

This approach, then, will not work. Instead, we must take a different approach. We will find a motive $F : (A + B) \times (A + B) \to \mathcal{U}$ such that:

$$\begin{aligned} F(\mathsf{inl}(a), \mathsf{inl}(a')) &\equiv \mathsf{Id}_A(a, a') \\ F(\mathsf{inr}(a), \mathsf{inr}(a')) &\equiv \mathsf{Id}_B(b, b') \\ F(\mathsf{inl}(a), \mathsf{inr}(b)) &\equiv 0 \\ F(\mathsf{inr}(a), \mathsf{inl}(b)) &\equiv 0 \end{aligned}$$

Such an F expresses all of the desired properties of the coproduct.

Exercise. *Define such an F by "double induction."*

The following lemma expresses the subgoal of our path induction with motive F:

Lemma. $x : A + B \vdash \_ : F(x, x)$.

Proof. Our proof of this will be a case statement:

$$\underline{\mathsf{case}}[z.F(z, z)](x; m : A.\mathsf{refl}_A(m), n : B.\mathsf{refl}_B(n)) : F(x, x)$$

Note that $\mathsf{refl}_A(m) : [\mathsf{inl}(m)/z]F(z, z)$, since

$$\mathsf{Id}_{A+B}(m, n) \equiv F(\mathsf{inl}(m), \mathsf{inl}(m)) \equiv [\mathsf{inl}(m)/z]F(z, z).$$

□

To complete the proof, we must define something of the type

$$\Pi x : A + B \Pi x' : A + B(\mathsf{Id}_{A+B}(x, y) \to F(x, x')).$$

This is our task for next time!

References

[1] N. G. de Bruijn. On the roles of types in mathematics. In Philippe de Groote, editor, *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de Logique*. Academia, 1995.

[2] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.

[3] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*, volume 1. Cambridge University Press, 1963. Reprint of the 1927 second edition. Accessed at `https://archive.org/details/PrincipiaMathematicaVolumeI`.

15-819 Homotopy Type Theory Lecture Notes

Joseph Lee and Kristina Sojakova

October 28 and 30, 2013

1 The Path Spaces of Coproducts

Recall from last week that we intend to characterize paths in coproducts by showing

$$\prod_{x:A+B} \prod_{x':A+B} \mathsf{Id}_{A+B}(x, x') \simeq F(x, x')$$

where $F : (A + B) \to (A + B) \to \mathcal{U}$ is defined by nested case-analysis so that

$$\begin{aligned} F(\mathsf{inl}(a), \mathsf{inl}(a')) &\equiv \mathsf{Id}_A(a, a') \\ F(\mathsf{inr}(a), \mathsf{inr}(a')) &\equiv \mathsf{Id}_B(b, b') \\ F(\mathsf{inl}(a), \mathsf{inr}(b)) &\equiv 0 \\ F(\mathsf{inr}(a), \mathsf{inl}(b)) &\equiv 0 \end{aligned}$$

To this end we define a function $f : \prod_{x:A+B} \prod_{x':A+B} \mathsf{Id}_{A+B}(x, x') \to F(x, x')$ by

$$f := \lambda x.\lambda x'.\lambda p.\ \mathsf{J}[F](p; z.\, \mathsf{case}(z; a.\mathsf{refl}_A(a); b.\mathsf{refl}_B(b)))$$

Next we need to define a function $g : \prod_{x:A+B} \prod_{x':A+B} F(x, x') \to \mathsf{Id}_{A+B}(x, x')$ such that $g(x, x')$ is a quasi-inverse of $f(x, x')$. We put

$$\begin{aligned} g := \lambda x.\lambda x'.\, \mathsf{case}(x; a.\, \mathsf{case}(x'; a'.\lambda z : F(\mathsf{inl}(a), \mathsf{inl}(a')).\mathsf{ap}_{\mathsf{inl}}(z); b'.\lambda z : F(\mathsf{inl}(a), \mathsf{inr}(b')).\mathsf{abort}_{(}z)); \\ b.\, \mathsf{case}(x'; a'.\lambda z : F(\mathsf{inr}(b), \mathsf{inl}(a')).\mathsf{abort}_{(}z); b'.\lambda z : F(\mathsf{inr}(b), \mathsf{inr}(b')).\mathsf{ap}_{\mathsf{inr}}(z))) \end{aligned}$$

We then have to exhibit terms

$$\begin{aligned} \alpha &: \prod_{x:A+B} \prod_{x':A+B} \prod_{u:F(x,x')} f(g(u)) =_{F(x,x')} u \\ \beta &: \prod_{x:A+B} \prod_{x':A+B} \prod_{v:\mathsf{Id}_{A+B}(x,x')} g(f(v)) =_{\mathsf{Id}_{A+B}(x,x')} v \end{aligned}$$

These terms are left as homework exercises.

Exercise. *Characterize the path space of the empty type* $\mathbf{0}$.

2 The Path Spaces of Identity Types

For a given type A, we would like to characterize the types $\mathsf{Id}_A(\_,\_)$, $\mathsf{Id}_{\mathsf{Id}_A(\_,\_)}(\_,\_)$, $\mathsf{Id}_{\mathsf{Id}_{\mathsf{Id}_A(\_,\_)}(\_,\_)}(\_,\_)$, and so on. While this is possible for certain specific types such as $\mathbf{0}, \mathbf{1}, A \times B, A \to B, A + B$ and Nat, it can be very difficult for other types, even seemingly simple ones. For example, determining the loop-spaces of n-spheres - i.e., path spaces based at a single point, denoted by $\Omega(S^n)$ - is a famous open problem in algebraic topology.

What we can say, however, is that if two types are equivalent then so are their path spaces:

Lemma. *If $f : A \to B$ is an equivalence, then so is $\mathsf{ap}_f : \mathsf{Id}_A(a, a') \to \mathsf{Id}_B(f(a), f(a'))$.*

Proof. Because f is an equivalence, it has a quasi-inverse $f^{-1} : B \to A$ and we have the following coherences:

- $\alpha : \prod_{a:A} f^{-1}(f(a)) =_A a$
- $\beta : \prod_{b:B} f(f^{-1}(b) =_B b$

In order to show that ap_f is an equivalence, it suffices to give a quasi-inverse $\mathsf{ap}_f^{-1} : \mathsf{Id}_B(f(a), f(a')) \to \mathsf{Id}_A(a, a')$, which we define by

$$\mathsf{ap}_f^{-1}(q) := \alpha(a)^{-1} \cdot \mathsf{ap}_{f^{-1}}(q) \cdot \alpha(a')$$

We now need to construct coherences

$$\gamma : \prod_{p:\mathsf{Id}_A(a,a')} \mathsf{ap}_f^{-1}(\mathsf{ap}_f(p)) =_{\mathsf{Id}_A(a,a')} p$$
$$\delta : \prod_{q:\mathsf{Id}_B(f(a),f(a'))} \mathsf{ap}_f(\mathsf{ap}_f^{-1}(q)) =_{\mathsf{Id}_B(f(a),f(a'))} q$$

This will imply that ap_f^{-1} is indeed a quasi-inverse of ap_f and thus both are equivalences We leave these as exercises. □

Exercise. *Define the coherence γ in the above proof by path induction.*

Exercise. *Define the coherence δ in the above proof as follows:*
1. Use the naturality of β to show that

$$\beta(f(a))^{-1} \cdot \mathsf{ap}_f(\mathsf{ap}_{f^{-1}}(q)) \cdot \beta(f(a')) =_{\mathsf{Id}_B(f(a),f(a'))} q$$

2. Use the naturality of α to show that

$$\alpha(f^{-1}(f(a)))^{-1} \cdot \mathsf{ap}_{f^{-1}}(\mathsf{ap}_f(\mathsf{ap}_{f^{-1}}(q))) \cdot \alpha(f^{-1}(f(a'))) =_{\mathsf{Id}_A(f^{-1}(f(a)),f^{-1}(f(a')))} \mathsf{ap}_{f^{-1}}(q)$$

3. Use the naturality of α to show that

$$\alpha(f^{-1}(f(a))) =_{\mathsf{Id}_A((f^{-1}(f(f^{-1}(f(a))))),f^{-1}(f(a)))} \mathsf{ap}_{f^{-1}}(\mathsf{ap}_f(\alpha(a)))$$

and similarly for a'.

4. Use 1), 2), 3) and the naturality of β to obtain the desired conclusion

$$\mathsf{ap}_f(\alpha(a)^{-1} \cdot \mathsf{ap}_{f^{-1}}(q) \cdot \alpha(a')) =_{\mathsf{Id}_B(f(a),f(a'))} q$$

3 Transport Properties of Identity

The identity type family on a type A can be thought of as a function $IdA(\_,\_) : A \to A \to \mathcal{U}$. Keeping the first argument $x : A$ fixed yields a type family $IdA(x,\_) : A \to \mathcal{U}$. For any path $q : y =_A y'$, the fibers $\mathsf{Id}_A(x,y)$ and $\mathsf{Id}_A(x,y')$ are related by the transport function

$$\mathsf{tr}[z.\mathsf{Id}_A(x,z)](q) : \mathsf{Id}_A(x,y) \to \mathsf{Id}_A(x,y')$$

Can we give an explicit description of this function? Clearly we can construct a function of the desired type "manually" by taking a $p : x = Ay$ and concatenating it with q to obtain $p \cdot q : x =_A y'$. Fortunately, it turns out this is precisely characterizes the behavior of the transport function, up to a propositional equality:

Lemma. *For any term $x : A$, and paths $q : y =_A y'$, $p : x =_A y$ we have*

$$\mathsf{tr}[z.\mathsf{Id}_A(x,z)](q)(p) =_{\mathsf{Id}_A(x,y')} p \cdot q$$

We have a similar characterization of transport in the case when the second argument is fixed:

Lemma. *For any term $y : A$, and paths $q : x =_A x'$, $p : x =_A y$ we have*

$$\mathsf{tr}[z.\mathsf{Id}_A(z,y)](q)(p) =_{\mathsf{Id}_A(x',y)} q^{-1} \cdot p$$

We notice that in the first case, we use the path q as-is whereas in the second case we first have to invert it. This can be described in category-theoretic terms as saying that the type family Id_A is *covariant in the second argument and contravariant in the first.*

Finally, we can consider the case of loops when the base point is allowed to vary:

Lemma. *For any paths $q : x =_A y$, $p : x =_A x$ we have*

$$\mathsf{tr}[z.\mathsf{Id}_A(z,z)](q)(p) =_{\mathsf{Id}_A(y,y)} q^{-1} \cdot p \cdot q$$

All these lemmas follow by straightforward path induction.

4 Justifying the Identity-Elimination Rule

Recall the identity elimination rule:

$$\frac{\Gamma \vdash P : \mathsf{Id}_A(M,N) \quad \Gamma, x{:}A, y{:}A, z{:}\mathsf{Id}_A(x,y) \vdash C\ \mathsf{type} \quad \Gamma, x{:}A \vdash Q : [x,x,\mathsf{refl}_A(x)/x,y,z]C}{\Gamma \vdash \mathsf{J}[x.y.z.C](P, x.Q) : [M,N,P/x,y,z]C}\ \mathsf{Id\text{-}E}$$

In Extensional Type Theory (ETT), this rule is a consequence of the identity reflection and UIP rules and thus has no special status.

In Intensional Type Theory (ITT), this rule can be understood as an induction principle: since the only way we can construct a proof of equality $P : M =_A N$ is by reflexivity in the case when $M \equiv N$, in order to prove $C[M,N,P]$ it is sufficient to prove $C[x,x,\mathsf{refl}_A(x)]$ for an arbitrary $x : A$. This intuition is justified by the following very important (and highly nontrivial) theorem:

Theorem 1. *In an empty context, two terms are propositionally equal if and only if they are definitionally equal and any identity proof is necessarily a reflexivity. In other words, if* $\vdash P : \mathsf{Id}_A(M,N)$*, then* $\vdash M \equiv N : A$ *and* $\vdash P \equiv \mathsf{refl}_A(M,M) : \mathsf{Id}_A(M,N)$*.*

In HoTT, it is no longer the case that every proof of equality is a reflexivity. For example, we have the following non-trivial identity proofs:

- $\mathsf{funext}(H) : f =_{A\to B} g$, where $H : \prod_{a:A} f(a) =_B g(a)$
- $\mathsf{ua}(E) : A =_{\mathcal{U}} B$, where $E : \sum_{f:A\to B} \mathsf{isequiv}(f)$
- $\mathsf{seq} : 0 =_{\mathbb{I}} 1$, where $\mathbb{I}$ is the interval type
- $\mathsf{loop} : b =_{S^1} b$, where S^1 is the circle type

This suggests that in HoTT, terms of an identity type should not be thought of purely as proofs of identity but rather as paths between terms. Since there can be potentially many distinct paths between two terms, the identity elimination rule should no longer be thought of as an induction principle.

The presence of nontrivial paths, however, poses a serious problem with the computational interpretation of HoTT: for example, what should $\mathsf{J}[\_](\mathsf{funext}(H); x.Q)$, $\mathsf{J}[\_](\mathsf{ua}(E))$, or $\mathsf{J}[\_](\mathsf{seq})$ compute to?

Even leaving aside univalence and higher inductive types, the addition of the function extensionality axiom to ITT poses a problem with computation. In ETT, we get function extensionality for free from the identity reflection rule. In Observational Type Theory (OTT), which combines intensional and extensional aspects, we get function extensionality with some special arrangements. Both ETT and OTT admit a computational interpretation, albeit by different means.

The computational interpretation of HoTT is currently a principal open problem. So how do we justify the J-rule as a suitable identity elimination rule? Given

- $C : \prod_{x:A} \prod y : A, \mathsf{Id}_A xy \to \mathcal{U}$
- $M, N : A$ and $P : M =_A N$
- $x : A \vdash Q : C[x, x, \mathsf{refl}_A(x)]$

why should there exist a term $\mathsf{J}[x.y.z.C](P; x.Q) : C[M, N, P]$?

Plugging M into Q, we obtain a term $Q[M] : C[M, M, \mathsf{refl}_A(M)]$. Similarly, plugging M into C yields a type family $C[M] : \prod_{y:A}(p : M =_A y) \to \mathcal{U}$, which can be equivalently understood as the function

$$\lambda z.C[M, \pi_1(z), \pi_2(z)] : \Big(\sum_{y:A} \mathsf{Id}_A(M, y)\Big) \to \mathcal{U}$$

Since functions are supposed to be functorial, constructing a path γ from $(M, \mathsf{refl}_A(M))$ to (N, P) in the type $\sum_{y:A} \mathsf{Id}_A(M, y)$ would give us a term

$$\mathsf{ap}_{\lambda z.C[M,\pi_1(z),\pi_2(z)]}(\gamma) : C[M, M, \mathsf{refl}_A(M)] =_{\mathcal{U}} C[M, N, P]$$

We could thus obtain our desired conclusion of the J-rule as

$$\mathsf{tr}[x : \mathcal{U}.x](\mathsf{ap}_{\lambda z.C[M,\pi_1(z),\pi_2(z)]}(\gamma))(Q[M]) : C[M, N, P]$$

In order to construct a path $\gamma : (M, \mathsf{refl}_A(M)) =_{\sum_{y:A} \mathsf{Id}_A(M,y)} (N, P)$, we need a characterization of path spaces of Σ-types, outlined in the following exercise:

Exercise. *Characterize the path space of the type $\Sigma_{x:A}B$ by constructing a term of type*

$$\prod_{p,p':\sum_{x:A} B(x)} (\mathsf{Id}_{\sum_{x:A} B(x)}(p, p') \simeq \sum_{q:\pi_1(p)=_A\pi_1(p')} \pi_2(p) =_q^{x.B} \pi_2(p'))$$

The above exercise tells us that in order to construct a path from $(M, \mathsf{refl}_A(M))$ to (N, P) in the type $\sum_{y:A} M =_A y$, it is sufficient to construct an element of the type $\sum_{q:M=_A N} \mathsf{refl}_A(M) =_q^{y.M=_A y} P$. The natural choice for the first component of the pair is the path $P : M =_A N$ itself. It thus remains to show that $\mathsf{refl}_A(M) =_P^{y.M=_A y} P$. In particular, this means showing that

$$\mathsf{tr}[y.M =_A y](P)(\mathsf{refl}_A(M)) =_{\mathsf{Id}_A(M,N)} P$$

By the first lemma in Section. 3, we have $\mathsf{tr}[y.M =_A y](P)(\mathsf{refl}_A(M)) =_{\mathsf{Id}_A(M,N)} \mathsf{refl}_A(M) \cdot P$. Since the left-hand evaluates to P, we are done.

5 Introduction to Homotopy Types

One way to see HoTT is that Homotopy *Type Theory* is *Homotopy Type* Theory. That is, HoTT can be thought of as the theory of homotopy types. We have already encountered several examples of the homotopy type *set*, also sometimes called an *h-set* or a *0-type*; however, we have not explicitly labeled them as such. We have the following definition:

Definition. *A type A is called a* set *(or a* 0-type*), if for all $x, y : A$ and $p, q : x =_A y$, we have that $p =_{\mathsf{Id}_A(x,y)} q$. In other words, the following type is inhabited:*

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

Intuitively, the type A can be viewed as "discrete up to homotopy". The familiar example of the type Nat of natural numbers (unsurisingly) turns out to be a set. More about this and other Homotopy Types next week!

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.

15-819 Homotopy Type Theory
Lecture Notes

Robert Lewis and Joseph Tassarotti

November 11 and 13, 2013

1 Contents

These notes cover Robert Harper's lectures on Homotopy Type Theory from November 11 and 13, 2013. Discussions include Hedberg's theorem, contractibility, propositional truncation, and the "axiom" of choice.

2 Refresher: Sets and Propositions

Recall from previous lectures the definitions of sets and propositions within HoTT. A type is called a set if there is only "one way" for any two of its elements to be equal:

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

Relatedly, a type is called a proposition if it is a "subsingleton": that is, it has at most one inhabitant.

$$\mathsf{isProp}(A) :\equiv \prod_{x,y:A} (x =_A y)$$

We can define isSet in terms of isProp: a type is a set if equality for that type is propositional.

$$\mathsf{isSet}(A) \equiv \prod_{x,y:A} \mathsf{isProp}(x =_A y)$$

What it means for a type to be a set is that there are no nontrivial relationships between elements of the type. The higher homotopy structure we

have seen in previous weeks does not exist in a set, as the only paths between elements are trivial, which trivializes the higher structure. In this sense, sets (and propositions) "reclaim" some properties of classical mathematics.

Proposition 1. *For any type A, we have* $\mathsf{isProp}(\neg A)$*, where* $\neg A :\equiv A \to 0$.

Proof. We want to find

$$\_ : \prod_{x,y:\neg A} (x =_{\neg A} y).$$

Since $\neg A$ is a function type, via funext it suffices to find

$$\_ : \prod_{u:A} (x(u) =_0 y(u)).$$

We have $\lambda u.\, \mathsf{abort}_{x(u)=_0 y(u)}(x(u))$ of this type. □

From this, we can derive the (perhaps surprising) result that $\neg\neg(\neg A) \to \neg A$, even though we do not necessarily have $\neg\neg A \to A$.

3 Hedberg's Theorem

These considerations lead us to the following important theorem.

Definition 1. *A type A has* decidable equality *if one can prove of any two inhabitants of A that they are either equal or unequal.*

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

A type A has stable equality *if double-negation elimination holds in its identity type:*

$$\prod_{x,y:A} (\neg\neg\mathsf{Id}_A(x,y) \to \mathsf{Id}_A(x,y))$$

Theorem 1. *A type with decidable equality is a set.*

Proof. The proof of Hedberg's theorem goes in two parts:

1. Decidable equality implies stable equality. In fact, we can prove in general that for any type A, $(A + \neg A) \to \neg\neg A \to A$. This part is simple and left as an exercise.
2. Stable equality implies sethood. This is the heart of Hedberg's theorem.

We prove 2. Suppose $h : \prod x, y : A.(\neg\neg x =_A y) \to (x =_A y)$ is evidence that equality in A is stable. To show $\mathsf{isSet}(A)$, is suffices to show that $x : A, p : x =_A x \vdash p =_{x=_A x} \mathsf{refl}_A(x)$, since we can reduce the identity of $p, q : x =_A y$ to showing that $p \cdot q^{-1} = \mathsf{refl}_A(x)$.

Fix $x : A$. We then have $h(x) : \prod y : A.(\neg\neg x =_A y) \to (x =_A y)$.

Using dependent function application apd (defined previously), we see that

$$\mathsf{apd}_{h(x)}(p) : p_*(h(x)(x)) =_{(\neg\neg x =_A x) \to (x =_A x)} h(x)(x)$$

By lemma 2.9.6 of [1], it follows that for that for any $r : \neg\neg(x =_A x)$,

$$p_*(h(x)(x))(r) =_{x=_A x} h(x)(x)(p_* r).$$

Next, from a proven property of transport in identity types, we have that

$$p_*(h(x)(x))(r) =_{x=_A x} h(x)(x)(r) \cdot p$$

and because negated types are propositions (from above),

$$h(x)(x)(p_* r) =_{x=_A x} h(x)(x)(r)$$

so we have by transitivity

$$h(x)(x)(r) \cdot p =_{x=_A x} h(x)(x)(r)$$

and cancellation gives us $p =_{x=_A x} \mathsf{refl}_A(x)$ as desired. □

For an example of the power of Hedberg's theorem, note that it implies $\mathsf{isSet}(\mathsf{Nat})$. Using double induction, one can show

$$\prod_{x,y:\mathsf{Nat}} (x =_{\mathsf{Nat}} y) \vee \neg(x =_{\mathsf{Nat}} y).$$

This is left as an exercise.

4 More Results on Propositions and Sets

Theorem 2. *Every proposition is a set:*

$$\text{If } \prod_{x,y:A} x =_A y \text{ then } \prod_{x,y:A} \prod_{p,q:x=_A y} p =_{x=_A y} q$$

Proof. Suppose $f : \prod x, y : A.x =_A y$. Given two inhabitants of A, f returns a path between them.

Fix $x_0 : A$ and let $g :\equiv f(x_0) : \prod y : A.x_0 =_A y$. For $p : y =_A y'$, we have

$$\mathsf{apd}_g(p) : p_*(g(y)) =_{x_0 =_A y'} g(y')$$

and by property of transport within identity type,

$$p_*(g(y)) =_{x_0 =_A y'} g(y) \cdot p.$$

By transitivity, we thus have

$$g(y) \cdot p =_{x_0 =_A y'} g(y')$$

$$p =_{x_0 =_A y'} g(y)^{-1} \cdot g(y)$$

For $q : y =_A y'$, these same calculations give us

$$q =_{x_0 =_A y'} g(y)^{-1} \cdot g(y).$$

Thus, $p =_{x_0 =_A y'} q$ as desired. □

Theorem 3. $\mathsf{isProp}(\mathsf{isProp}(A))$ – *that is, there is only one proof that A has only one inhabitant.*

Proof. Let $f, g : \mathsf{isProp}(A)$ be given. To show $f =_{\mathsf{isProp}(A)} g$, it suffices to show (by funext) $x, y : A \vdash \_ : fxy =_{x =_A y} gxy$. This follows since $\mathsf{isProp}(A) \to \mathsf{isSet}(A)$. □

We can similarly prove $\mathsf{isProp}(\mathsf{isSet}(A))$. For $f : A \to B$, with the proper notion of $\mathsf{isEquiv}(f)$, we will soon be able to prove $\mathsf{isProp}(\mathsf{isEquiv}(f))$. If we take equivalence to mean having a quasi-inverse, though, this is not the case.

5 Contractibility

In preparation for what follows, we define the notion of contractibility:

$$\mathsf{isContr}(A) : \sum_{x:A} \prod_{y:A} x =_A y$$

That is, a type is contractible if there is some inhabitant which all other inhabitants are equal to. This is equivalent to saying that A is a prop, and it is inhabited. Alternatively, a A type is contractibile if it is equivalent to 1.

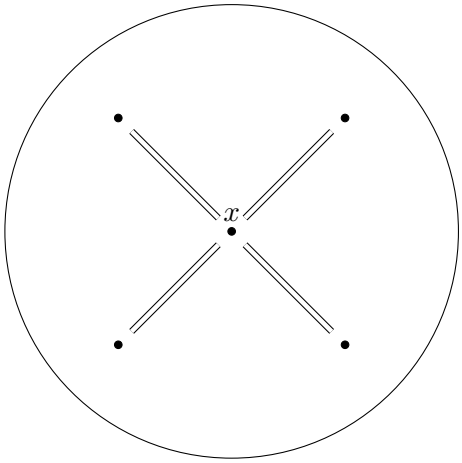


Figure 1: A contractible type. There is some element x, and paths between x and all other elements

Notice that for any type A, given $a{:}A$, we have that $\mathsf{isContr}(\Sigma x{:}A a = x)$. In particular, $(a, \mathsf{refl}_A(a)))$ is an inhabitant of this type which all other elements are equal to.

The notions of contraction and truncation are related to the (n)-type hierarchy: specifically, contractions sit at the bottom of the hierarchy. For historical reasons, we begin with (-2) types, and define inductively:

$$\begin{aligned} \mathsf{is\text{-}}(-2)\mathsf{\text{-}type}(A) &:\equiv \mathsf{isContr}(A) \\ \mathsf{is\text{-}}(n+1)\mathsf{\text{-}type}(A) &:\equiv \prod_{x,y:A} \mathsf{is\text{-}}n\mathsf{\text{-}type}(x =_A y) \end{aligned}$$

Now, we have that:

$$\mathsf{isProp}(A) \leftrightarrow \prod_{x,y:A} \mathsf{isContr}(x =_A y).$$

which implies that

$$\mathsf{isProp}(A) \leftrightarrow \mathsf{is\text{-}}(-1)\mathsf{\text{-}type}(A).$$

We can further prove the following:

$$\begin{aligned} \mathsf{isSet}(A) &\leftrightarrow \mathsf{is\text{-}0\text{-}type}(A) \\ \mathsf{isGpd}(A) &\leftrightarrow \mathsf{is\text{-}1\text{-}type}(A) \\ \mathsf{is\text{-}2\text{-}Gpd}(A) &\leftrightarrow \mathsf{is\text{-}2\text{-}type}(A) \\ &\vdots \end{aligned}$$

As well as

$$\prod_{A:\mathcal{U}} (\mathsf{is\text{-}}n\mathsf{\text{-}type}(A) \to \mathsf{is\text{-}}(n+1)\mathsf{\text{-}type}(A)) .$$

However, it is *not* the case that every type is an n-type for some n: consider $\mathcal{U}$.

6 Propositional Truncation (Squashing)

The notion of "squashing" introduced last week was perhaps too heavy-handed: it was used toward a number of goals, among them to recover classical logic within constructive logic. We now introduce the more general idea of *abstract truncation*, which for now will be taken as a primitive idea of HoTT. Truncation serves to reduce types to sets, without all the byproducts of the double negation translation.

Let $\|A\|_{-1}$ be read as the *(-1)-truncation* of A. When the context is clear, we omit the subscript. When the type $\|A\|$ is inhabited, we say A is "merely inhabited", to emphasize that this is a proof-irrelevant setting. We have the following $\|\cdot\|$-introduction rules:

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash |M| : \|A\|} \qquad \frac{}{\Gamma, x : \|A\|, y : \|A\| \vdash \mathsf{squash}(x, y) : \mathsf{Id}_{\|A\|}(x, y)}$$

As we would expect, $\mathsf{isProp}(\|A\|)$ because of the rule for $\mathsf{squash}(\cdot,\cdot)$. The corresponding elimination rule s:

$$\frac{\Gamma \vdash M : \|A\| \quad \Gamma, x : A \vdash N : B \quad \Gamma \vdash P : \mathsf{isProp}(B)}{\mathsf{elim}[B](M; x.N; P) : B}\ \|\cdot\|E$$

We require a proof of $\mathsf{isProp}(B)$ to ensure that the behavior of N does not depend on the representative $x : A$. There are other ways to ensure

this property: for instance, we could instead require $u, v : A \vdash [u/x]N =_B [v/x]N$. Note that $\mathsf{isProp}(B)$ implies that this is the case.

The β rule works as one might expect:

$$\mathsf{elim}(|M|; x.N; P) \equiv [M/x]N$$

One would similarly like some β like rule to hold for the 1-cell, squash. For example, something of the form

$$\mathsf{ap}(\lambda z.\mathsf{elim}(z; x.N; P))(\mathsf{squash}(|M|, |M'|)) \equiv P([M/x]N)([M'/x]N)$$

which corresponds to the idea that P is a proof that N is equal under substitutions of different terms of type A, because B is a proposition. However, just as in the case of seg, we do not have this.

7 Revisiting the Axiom of Choice

Previously, we explored how the axiom of choice is provable in ITT. That is, there is a term AC_∞ such that

$$AC_\infty : \prod_{A:U} \prod_{B:A\to U} \prod_{C:\Pi x:A.B\to U} \left(\left(\prod_{x:A} \sum_{y:B(x)} C(x,y) \right) \to \left(\sum_{f:\Pi x:A.B} \prod_{x:A} C(x, f(x)) \right) \right)$$

In fact, we can strengthen this and say that the two types are equivalent. Recall that this type is inhabited because the witness that the relation C is total provides precisely the choice we should make, because the specification is proof relevant. The situation is similar to the example that motivated our introduction of propositional truncation, where we wanted to write a total function that returned the index of the first occurrence of 0 in an infinite sequence. The proof that the infinite sequence actually contained a 0 also immediately told us where the 0 was.

Now that we have developed the notion of propositional truncation, we can state a version of the axiom of choice that is closer in meaning to its typical statement:

$$\prod_{A:U} \prod_{B:A\to U} \prod_{C:\Pi x:A.B\to U} \left(\mathsf{isSet}(A) \to \left(\prod_{x:A} \mathsf{isSet}(B(x)) \right) \to \left(\prod_{x:A} \prod_{y:B(x)} \mathsf{isProp}(C(x,y)) \right) \right.$$
$$\left. \to \left(\left(\prod_{x:A} \left\| \sum_{y:B_x} C(x,y) \right\| \right) \to \left\| \sum_{f:\Pi x:A.B} \prod_{x:A} C(x, f(x)) \right\| \right) \right)$$

This restated form is not provable. What has happened is now that we say that there merely exists some y for each x such that $C(x, y)$. The axiom says that given such weaker evidence, there merely exists such a function f where for each x, $C(x, f(x))$. We might call an axiom of such a type AC_{-1}, to emphasize that it involves the -1-truncation.

Now, the truncations of equivalent types are equivalent. Since the non-truncated form of the axiom of choice, AC_∞, gives us the equivalence

$$\left(\left(\prod_{x:A}\sum_{y:B(x)} C(x,y)\right) \simeq \left(\sum_{f:\Pi x:A.B}\prod_{x:A} C(x,f(x))\right)\right)$$

so that the type of AC_{-1} is equivalent to

$$\prod_{A:U}\prod_{B:A\to U}\prod_{C:\Pi x:A.B\to U}\left(\mathsf{isSet}(A) \to \left(\prod_{x:A}\mathsf{isSet}(B(x))\right) \to \left(\prod_{x:A}\prod_{y:B(x)}\mathsf{isProp}(C(x,y))\right)\right.$$
$$\left.\to \left(\left(\prod_{x:A}\left\|\sum_{y:B_x} C(x,y)\right\|\right) \to \left\|\prod_{x:A}\sum_{Y:B(x)} C(x,y)\right\|\right)\right)$$

Now, for all $Y : A \to U$, we have that $\Pi_{x:A}Y(x) \simeq \Pi_{x:A}(\Sigma_{a:Y(x)}1)$, so we can simplify the above type to:

$$\prod_{A:U}\prod_{Y:A\to U}\left(\mathsf{isSet}(A) \to \left(\prod_{x:A}\mathsf{isSet}(Y(x))\right) \to \left(\left(\prod_{x:A}||Y(x)||\right) \to \left\|\prod_{x:A}Y(x)\right\|\right)\right)$$

In this form, we see that the axiom is saying that a product of a family of merely inhabited sets is merely inhabited, which is well-known to be equivalent to the axiom of choice in classical mathematics. It is crucial here that $\mathsf{isSet}(A)$, since if A is not a set, there is a counter example (see lemma 3.8.5 in [1])

8 Equivalence and Propositions

A few weeks ago, we defined what it meant for a map $f : A \to B$ to be an equivalence between A and B, written $\mathsf{isequiv}(f)$. The definition we gave was:

$$\mathsf{isequiv}(f) :\equiv (\Sigma g : B \to A.f \circ g \sim \mathsf{id}_B) \times (\Sigma h : B \to A.h \circ f \sim \mathsf{id}_A).$$

We also gave the related notion of a quasi-inverse, written $\mathsf{qinv}(f)$, which was defined as:

$$\mathsf{qinv}(f) :\equiv \Sigma g : B \to A.(f \circ g \sim \mathsf{id}_B \times g \circ f \sim \mathsf{id}_A).$$

In some ways, the definition of qinv may at first appear to be a more natural definition than isequiv, since it states that there is some function g which is a left and right inverse of f. This is the definition of an isomorphism in category theory, for example. Of course, we explained that $\mathsf{isequiv}(f) \to \mathsf{qinv}(f)$ and $\mathsf{qinv}(f) \to \mathsf{isequiv}(f)$, and this enabled us to use whatever definition was more convenient over the past few weeks.

Why did we choose the above definition for isequiv instead of using the definition we gave for qinv? The issue is that we would like there to be only one proof that a given function f is an equivalence. That is, we want $\mathsf{isProp}(\mathsf{isequiv}(f))$ to hold for all f. This is the case for the definition we gave, but it is not true that $\mathsf{isProp}(\mathsf{qinv}(f))$ for every f. We can show this by establishing two lemmas:

Proposition 2. *If* $f : A \to B$ *and* $e : \mathsf{qinv}(f)$ *then* $\mathsf{qinv}(f) \simeq \Pi x{:}A.(x = x)$

Proposition 3. *There exists some type* X *such that* $\Pi x{:}X(x = x)$ *is not a proposition.*

See the discussion in section 4.1 in [1] for proofs of these lemmas. This makes qinv unsuitable as a definition for isequiv. Nevertheless, we would still like a definition of isequiv to be interprovable with qinv, while also being a proposition. There are three candidates which satisfy these properties, all of which are equivalent:

1. $\mathsf{biequiv}(f) :\equiv (\Sigma g : B \to A.f \circ g \sim \mathsf{id}_B) \times (\Sigma h : B \to A.h \circ f \sim \mathsf{id}_A)$.
   We say that f is *bi-invertible*, which means that f has a left inverse and a right inverse. This is the definition we have been using.
2. $\mathsf{isContr}(f) :\equiv \Pi y{:}B.\, \mathsf{isContr}(\mathsf{fib}_f(y))$ where $\mathsf{fib}_f(y) = \Sigma x{:}A.f(x) = y$.
   This says that f is contractible if given any y in the codomain, the set of all things that f maps to y (the fiber), is contractible. That is, for every point in the codomain, there is an element x in the domain such that $f(x) = y$, and if $f(x') = y$ then $x = x'$. But that precisely means that f is a bijection up to homotopy.
3. $\mathsf{ishae}(f) :\equiv \Sigma g{:}B \to A.\Sigma\alpha{:}(f \circ g \sim id).\Sigma\beta{:}(g \circ f \sim id).$
   $\Pi x{:}\ A.f(\beta x) = \alpha(fx)$:
   We read this as saying that f is a *half-adjoint equivalence*. We did not talk about this definition in class, but a discussion can be found in section

4.2 of [1]. Roughly, we can motivate this by noticing that it is similar to the definition of qinv (where the homotopies α and β were unnamed) with an additional coherence condition relating how these homotopies interact with f.

References

[1] Institute for Advanced Study. *Homotopy Type Theory: Univalent Foundations of Mathematics*. The Univalent Foundations Program, 2013. `http://homotopytypetheory.org/book/`.

15-819 Homotopy Type Theory
Lecture Notes

Evan Cavallo and Stefan Muller

November 18 and 20, 2013

1 Reconsider Nat in simple types

As a warmup to discussing inductive types, we first review several equivalent presentations of the simple type Nat seen earlier in the course. The introduction forms for Nat are 0 and $\mathsf{succ}(M)$ for any $M : \mathsf{Nat}$. The elimination form is the recursor rec.

1.1 Traditional Form

$$\frac{}{\Gamma \vdash 0 : \mathsf{Nat}}\ \mathsf{Nat}I_{z1} \qquad\qquad \frac{\Gamma \vdash M : \mathsf{Nat}}{\Gamma \vdash \mathsf{succ}(M) : \mathsf{Nat}}\ \mathsf{Nat}I_{s1}$$

$$\frac{\Gamma \vdash M : \mathsf{Nat} \qquad \Gamma \vdash M_0 : A \qquad \Gamma, x : A \vdash M_1 : A}{\Gamma \vdash \mathsf{rec}[A](M; M_0; x.M_1) : A}\ \mathsf{Nat}E_1$$

We include the motive A in the recursor to motivate the dependently-typed presentation to come although it is not necessary in the simply-typed setting. The dynamic behavior of rec is defined by the following β rules.

$$\mathsf{rec}[A](0; M_0; x.M_1) \equiv M_0$$

$$\mathsf{rec}[A](\mathsf{succ}(M); M_0; x.M_1) \equiv [\mathsf{rec}[A](M; M_0; x.M_1)/x]M_1$$

The recursor on 0 returns the base case M_0. On $\mathsf{succ}(M)$, it substitutes the recursive result on M for x in M_1. The η rule states that any object that "behaves like" the recursor is definitionally equal to the recursor on the appropriate arguments.

$$\frac{[0/y]N \equiv M_0 \qquad \Gamma, z : \mathsf{Nat} \vdash [\mathsf{succ}(z)/y]N \equiv [[z/y]N/x]M_1 : A}{\Gamma, y : \mathsf{Nat} \vdash N \equiv \mathsf{rec}[A](y; M_0; x.M_1)}\ \eta$$

1.2 As elements of the exponential

The introduction forms of Nat may be treated as exponentials in the absence of a context.

$$\cdot \vdash 0 : 1 \to \mathsf{Nat} \qquad (\mathsf{Nat}I_{z3})$$

$$\cdot \vdash \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat} \qquad (\mathsf{Nat}I_{s3})$$

Note that the type $1 \to \mathsf{Nat}$ is equivalent to the type Nat. There are two ways to present the elimination form in this format. The first moves the Nat on which the recursion is done to the argument position, implying that rec has exponential type.

$$\frac{\cdot \vdash M_0 : A \qquad x : A \vdash M_1 : A}{z : \mathsf{Nat} \vdash \mathsf{rec}[A](M_0; x.M_1)(z) : A}\ \mathsf{Nat}E_{3a}$$

This can be presented in a more direct way by omitting the argument.

$$\frac{\cdot \vdash M_0 : A \qquad x : A \vdash M_1 : A}{\cdot \vdash \mathsf{rec}[A](M_0; x.M_1) : \mathsf{Nat} \to A}\ \mathsf{Nat}E_{3b}$$

We can derive rules $\mathsf{Nat}I_{z1}$, $\mathsf{Nat}I_{s1}$ and $\mathsf{Nat}E_1$. For example,

$$\frac{\dfrac{}{\Gamma, M : \mathsf{Nat} \vdash M : \mathsf{Nat}} \qquad \dfrac{}{\Gamma, M : \mathsf{Nat} \vdash \mathsf{succ} : \mathsf{Nat} \to \mathsf{Nat}}}{\Gamma, M : \mathsf{Nat} \vdash \mathsf{succ}(M) : \mathsf{Nat}}\ \mathsf{Nat}I_{s3}$$

2 Nat-algebras

We now motivate the idea of Nat-algebras, which are maps of the form $1 + A \to A$. From the name, one would expect there to be a Nat-algebra where A is Nat. Indeed, there is.

$$z : 1 + \mathsf{Nat} \vdash \mathsf{case}(z; \_.0; x.\mathsf{succ}(x)) : \mathsf{Nat}$$

We can write $\mathsf{case}(z; \_.0; x.\mathsf{succ}(x))$ above as $\{\_.0; x.\mathsf{succ}(x)\}(z)$ or, somewhat abusively, $\{0, \mathsf{succ}\}(z)$. This gives

$$\cdot \vdash \{0, \mathsf{succ}\} : 1 + \mathsf{Nat} \to \mathsf{Nat}$$

as desired. More generally, we can write any Nat-algebra as $\alpha = \{\alpha_0, \alpha_1\}$ where $\alpha_0 : 1 \to \mathsf{Nat}$ (or, equivalently, $\alpha_0 : \mathsf{Nat}$) and $\alpha_1 : \mathsf{Nat} \to \mathsf{Nat}$. We call α_0 the *basis* or *pseudo-zero* and α_1 the *inductive step* or *pseudo-successor*.

In fact, $\{0, \mathsf{succ}\}$ holds a special position among Nat-algebras. The Nat-algebras form a category and $\{0, \mathsf{succ}\}$ is the initial object in this category. Recall that this means it has a unique morphism to any other object in the category. This requires us to define

morphisms between Nat-algebras, which we will call Nat-*homomorphisms*. Given two Nat-algebras, $\alpha : 1 + A \to A$ and $\beta : 1 + B \to B$, $h : A \to B$ is a Nat-homomorphism if it makes the following diagram commute.

$$\begin{array}{ccc} 1 + A & \xrightarrow{1+h} & 1 + B \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{\quad h \quad} & B \end{array}$$

The map $1 + h : 1 + A \to 1 + B$ is defined in the natural way:

$$1 + h :\equiv \{\_.\mathsf{inl}\ \langle\rangle; a.\mathsf{inr}\ h(a)\}$$

To show that Nat is an initial algebra, we must show that for every Nat-algebra $\alpha : 1 + A \to A$, there exists a unique Nat-homomorphism $! : \mathsf{Nat} \to A$ such that the following diagram commutes (note that the order of quantifications expressed in the previous sentence is not inherently clear in the diagram, and must be considered to get a full understanding of the diagram.)

$$\begin{array}{ccc} 1 + \mathsf{Nat} & \overset{1+!}{\dashrightarrow} & 1 + A \\ \downarrow{\scriptstyle \{0,\mathsf{succ}\}} & & \downarrow{\scriptstyle \alpha} \\ \mathsf{Nat} & \overset{!}{\dashrightarrow} & A \end{array}$$

Let's consider the requirements on ! for the diagram to commute.

$$!0 = \alpha_0$$

$$!\mathsf{succ}(x) = \alpha_1(!x)$$

where the left sides correspond to following the path $! \circ \{0, \mathsf{succ}\}$ and the right sides correspond to following $\alpha \circ 1{+}!$. Note that these two equations match the β rules for rec, so we can define $! :\equiv \mathsf{rec}[A](\alpha_0; \alpha_1)$ or simply $! :\equiv \mathsf{rec}[A](\alpha)$. As we see above, the β rules for rec imply commutation of the diagram. Uniqueness of ! follows from the η rule for rec.

It's worth noting that commuting diagrams hide exactly the type of equality that is being discussed, which is quite important in HoTT. For Nat, for example, uniqueness of ! holds "on the nose," while, in general, uniqueness may be only up to higher homotopy.

3 F-algebras

The above discussion can be generalized to any functor F. A functor is a mapping between categories C and D. Functors act on objects in a category and the morphisms between

them, i.e. for all objects $X \in C$, $F(X) \in D$, and for all morphisms $f : X \to Y$ between objects X and Y in C, $F(f) : F(X) \to F(Y)$. Functors respect identity and composition:
1. For every object $X \in C$, $F(\mathsf{id}_X) = \mathsf{id}_{F(X)}$
2. For all morphisms $f : X \to Y$ and $g : Y \to Z$, $F(g \circ f) = F(g) \circ F(f)$
[1]

For example, $F_{\mathsf{Nat}}(C) :\equiv 1 + C$ is a functor. We check that F_{Nat} preserves identities and composition. Let X be an object of C.

$$F_{\mathsf{Nat}}(\mathsf{id}_X) = 1 + \mathsf{id}_X = \{\langle\rangle, \mathsf{id}_X\}$$

which is indeed an identity on $1 + X$. Let $f : X \to Y, g : Y \to Z$ be morphisms between objects of C.

$$F_{\mathsf{Nat}}(g \circ f) = 1 + g \circ f = \{\langle\rangle, g \circ f\} = \{\langle\rangle, g\} \circ \{\langle\rangle, f\} = (1 + g) \circ (1 + f) = F(g) \circ F(f)$$

For any functor F, an F-algebra is a mapping $F(X) \to X$. Thus, a Nat-algebra is an F_{Nat}-algebra. F-algebras form categories as Nat-algebras do. For a functor F, objects A and B, two F-algebras $\alpha : F(A) \to A$ and $\beta : F(B) \to B$, and a morphism $h : A \to B$, the following diagram commutes.

$$\begin{array}{ccc} F(A) & \xrightarrow{F(h)} & F(B) \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle \beta} \\ A & \xrightarrow{h} & B \end{array}$$

An *initial* F-algebra is an F-algebra $i : F(I) \to I$ such that for all other F-algebras $\alpha : F(A) \to A$, there exists a unique map $! : I \to A$ such that the following diagram commutes.

$$\begin{array}{ccc} F(I) & \overset{F(!)}{\dashrightarrow} & F(A) \\ \downarrow{\scriptstyle i} & & \downarrow{\scriptstyle \alpha} \\ I & \overset{!}{\dashrightarrow} & A \end{array}$$

There also exists the notion of an F-coalgebra, which, dual to the above, is a map $\alpha : A \to F(A)$. A final F-coalgebra is a mapping $j : J \to F(J)$ such that for all other F-coalgebras $\alpha : A \to F(A)$, there exists a unique map $! : A \to J$ making the following diagram commute.

$$\begin{array}{ccc} A & \overset{!}{\dashrightarrow} & J \\ \downarrow{\scriptstyle \alpha} & & \downarrow{\scriptstyle j} \\ F(A) & \overset{!}{\dashrightarrow} & F(J) \end{array}$$

Lemma 1 (Lambek)**.** *If $i : F(I) \to I$ is an initial F-algebra, then i is an isomorphism. That is, $F(I) \equiv I$.*

Proof. To show that i is an isomorphism, we must exhibit an inverse $i^{-1} : I \to F(I)$ such that $i \circ i^{-1} = \mathsf{id}_I$ and $i^{-1} \circ i = \mathsf{id}_{F(I)}$. Consider the F-algebra $F(i) : F(F(I)) \to F(I)$. We now treat i as homomorphism between the F-algebras $F(i)$ and i making this diagram commute.

$$\begin{array}{ccc} F(F(I)) & \xrightarrow{F(i)} & F(I) \\ \downarrow{\scriptstyle F(i)} & & \downarrow{\scriptstyle i} \\ F(I) & \xrightarrow{\quad i \quad} & I \end{array}$$

Since i is an initial F-algebra, however, we also have a unique mapping $! : I \to F(I)$ making the top half of this diagram commute.

$$\begin{array}{ccc} F(I) & \xrightarrow{\quad i \quad} & I \\ \vdots{\scriptstyle F(!)} & & \vdots{\scriptstyle !} \\ F(F(I)) & \xrightarrow{F(i)} & F(I) \\ \downarrow{\scriptstyle F(i)} & & \downarrow{\scriptstyle i} \\ F(I) & \xrightarrow{\quad i \quad} & I \end{array}$$

There is also a unique mapping from I to I, which must be the identity. This indicates that the mapping $i \circ !$ along the right side of the diagram must be equal to id_I:

$$i \circ ! = \mathsf{id}_I$$

We also have

$$! \circ i = F(i) \circ F(!) = F(i \circ !) = F(\mathsf{id}_I) = \mathsf{id}_{F(I)}$$

where the first equality follows from the commutativity of the upper half of the diagram, the second and fourth follow from the properties of functors and the third follows from the result above. □

This shows that for any functor F, the initial F-algebra I is a fixed point of F. A dual result can be proven showing that, if J is the final F-coalgebra of a functor F, then $F(J) \equiv J$.

4 Internalizing Nat-Algebras

Inside type theory, we can define the notion of Nat-algebra as

$$\mathsf{NatAlg} :\equiv \Sigma A{:}\mathcal{U}.\, ((1 + A) \to A)$$

We can define the type of Nat-homomorphisms between two Nat-algebras (A, α) and (B, β) as

$$\mathsf{NatHom}(\alpha, \beta) :\equiv \Sigma h{:}A \to B.\, (\beta \circ (1 + h) = h \circ \alpha)$$

The fact that $\nu :\equiv (\mathsf{Nat}, \{0, \mathsf{succ}\})$ is initial in the category of Nat-algebras is expressed by the fact that $\mathsf{NatHom}(\nu, \alpha)$ is contractible for all α: this means that there exists a Nat-homomorphism from ν to α which is unique up to higher homotopy.

5 W-types

We'd like to be able to take a functor F and define the initial F-algebra within HoTT (if one exists). For the class of *polynomial functors*, we can do this using *Brouwer ordinals*, also called W-*types*.

W-types are inspired by the mathematical concept of well-founded induction. In classical mathematics, a partially ordered set $\langle A, < \rangle$ is said to be well-founded if every subset of A has a $<$-minimal element (equivalently, there are no infinite descending chains). Well-founded sets are useful because they admit an induction principle:

> **Proposition (Well-Founded Induction):**
> Let $\langle A, < \rangle$ be a well-founded set and $P(x)$ be a proposition. If for any $a \in A$ we can prove $P(a)$ by assuming $P(b)$ for all $b < a$, then $P(a)$ holds for all $a \in A$.

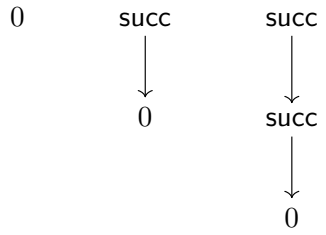
The set of natural numbers $\langle \mathbb{N}, < \rangle$ together with its usual ordering is an example of a well-founded set, and the induction principle is the familiar mathematical induction.

Classically, the proof that induction holds goes by contradiction, so this definition is unsatisfactory for a constructive theory. We will instead characterize well-founded sets as those for which we have a (constructive) induction principle.

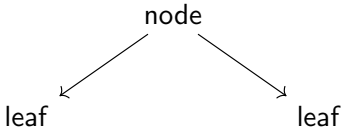
To better understand what we mean by this, we will define W-types. To form a W-type, we require a type A and a type family B over A:

$$\frac{\Gamma \vdash A : \mathcal{U} \quad \Gamma, x{:}A \vdash B : \mathcal{U}}{\Gamma \vdash \mathsf{W}x{:}A.B : \mathcal{U}}\ \mathsf{W}F$$

A is the type of *node sorts*. Each node sort represents a different way of forming an element of $\mathsf{W}x{:}A.B$. For example, in the case of the natural numbers, the node sorts are 0 and succ. We can think of each element of the natural numbers as a tree built from these two sorts of nodes. For example, the numbers 0 through 2 can be represented as



Another natural example is the type of binary trees. This type can be defined by two node sorts node and leaf, with the elements taking the form of trees such as this:



In order to fully specify these two types, we need to have some notion of a node's arity. This is given by the type family B. For each node sort $a : A$, $B(a)$ describes the *branching factor* of a, the index type to specify the predecessors of a node of sort a. For example, the branching factor of the sort 0 or leaf would be 0, the branching factor of succ would be 1, and the branching factor of node would be $1+1$. Thus we can write Nat as $\mathsf{W}x{:}2.\,\mathsf{if}(x; 0; 1)$, where tt represents 0 and ff represents succ.

Now that we have the purposes of A and B in hand, we can see how to define the introduction rule for $\mathsf{W}x{:}A.B$.

$$\frac{a : A \quad x{:}B(a) \vdash w : \mathsf{W}x{:}A.B}{\Gamma \vdash \mathrm{sup}[a](x.w) : \mathsf{W}x{:}A.B}\ \mathsf{W}I$$

In other words, in order to construct a node of sort a, we must give an element $\mathsf{W}x{:}A.B$ for each predecessor as specified by the branching factor $B(a)$. Note that when $B(a)$ is 0 we can construct a new node without any predecessor information. For example, we can construct elements of the naturals as follows:

$$\begin{aligned}
\overline{0} &:\equiv \mathrm{sup}[\mathsf{tt}](x.\mathsf{abort}_{\mathsf{W}x:2.\,\mathsf{if}(x;0;1)}(x)) \\
\overline{1} &:\equiv \mathrm{sup}[\mathsf{ff}](_.\overline{0}) \\
\overline{2} &:\equiv \mathrm{sup}[\mathsf{ff}](_.\overline{1})
\end{aligned}$$

The recursor for $\mathsf{W}x{:}A.B$ follows the idea of well-founded recursion: in order to define the result of a function f on an element $w : \mathsf{W}x{:}A.B$, we can assume we've already computed f for all of w's predecessors.

$$\frac{\Gamma \vdash C : \mathcal{U} \quad \Gamma, a : A, r : B(a) \to C \vdash M : C}{\Gamma, z : \mathsf{W}x{:}A.B \vdash \mathsf{wrec}[C](a, r.M)(z) : C}\ \mathsf{W}R$$

In the hypothesis, we assume that we are dealing with a node of type a and that we have the value $r(b)$ for each $b : B(a)$ indexing a predecessor. We use this information to construct the value M of the recursor at the current node. The recursor comes with the compuation rule

$$\mathsf{wrec}[C](a, r.M)(\sup[a](w)) \equiv [a, \lambda z.\mathsf{wrec}[C](a, r.M)(w(z))/a, r]M$$

which, as expected, gives the value of $\mathsf{wrec}[C](a, r.M)$ on $\sup[a](w)$ in terms of the value on each predecessor $w(z)$ for $z : B(a)$.

The dependent eliminator has a similar form, expressing the idea of well-founded induction.

$$\frac{\Gamma, z : \mathsf{W}x{:}A.B \vdash P : \mathcal{U} \quad \Gamma, a{:}A, p : B(a) \to \mathsf{W}x{:}A.B, h : \prod_{b:B(a)} P(p(b)) \vdash M : P(\sup[a](p))}{\Gamma, z : \mathsf{W}x{:}A.B \vdash \mathsf{wind}[x.P](a, p, h.M) : P(z)}\ \mathsf{W}E$$

Here, in order to formulate the hypothesis, we need to assume the additional data $p : B(a) \to \mathsf{W}x{:}A.B$ which gives us the predecessors of the element we are considering. The computation rule takes the form

$$\mathsf{wind}[x.P](a, p, h.M)(\sup[a](w)) = [a, w, \lambda z.\mathsf{wind}[x.P](a, p, h.M)(w(z))/a, p, h]M$$

In general, we can only assert that this computation rule holds propositionally.

Each W-type determines a functor, in particular a *polynomial functor*. This is a functor of the form $F(X) = \Sigma a{:}A.(B(a) \to X)$ for some type A and type family B. We can see that the W-type $\mathsf{W}x{:}A.B$ defines an F-algebra where $F(X) :\equiv \Sigma a{:}A.(B(a) \to X)$: we have $\lambda(a, w).\sup[a](w) : F(X) \to X$. The map $\lambda(a, w).\sup[a](w)$ is in fact an equivalence, and $\mathsf{W}x{:}A.B$ is a homotopy-initial F-algebra.

In the case of our W-type definition of Nat, observe that the functor determined by $\mathsf{W}x{:}\mathsf{2}.\,\mathsf{if}(x; 0; 1)$ is $F(X) = \Sigma b{:}\mathsf{2}.(\mathsf{if}(b; 0; 1) \to X)$. One can check that $\Sigma b{:}\mathsf{2}.(\mathsf{if}(b; 0; 1) \to X) \simeq 1 + X$. Thus, this type satisfies the equation $F(X) = 1 + X$, our original definition of a Nat-algebra.

References

[1] Wikipedia. Functor. `http://en.wikipedia.org/wiki/Functor`, 2013.

15-819 Homotopy Type Theory
Lecture Notes

Kristina Sojakova and Joseph Lee

November 25, 2013

1 Higher Inductive Types

Recall last week we discussed (lower) inductive types and their definitions. This week, we move on to discussing higher inductive types. Intuitively, a higher inductive type (i.e. HIT) can be seen as a type with inductive definitions along with equational laws.

A higher order type then is a generalization of a free algebraic structure, which would have generators and equational laws that should be fulfilled. For example, a group given by a set of generators and relations (e.g. commutativity).

This gives a full higher-dimensional structure which allows us to impose relations in multiple dimensions. In a proof-relevant environment, this means we get generators and more generators. Or in other words, because of proof-relevance, equational laws can be seen as generators at a higher dimension.

For example, at the 0-type level, the generators are elements or points. These are called the 0-cells. Identities between 0-cells would be 1-cells, identities between 1-cells would be 2-cells, and so on. This allows for a type to be defined over multiple dimensions by the use of n-cells.

2 The Interval Type I

Recall that the interval type I is defined by two points, 0_I and 1_I, along with an identity seg. That is, I is defined by the 0-cells

$$0_I : I$$
$$1_I : I$$

and the 1-cell

$$\mathsf{seg} : 0_I =_I 1_I$$

The recursor is defined as

$$\frac{\Gamma \vdash M : I \quad \Gamma \vdash a : A \quad \Gamma \vdash b : A \quad \Gamma \vdash \beta : a =_A b}{\Gamma \vdash \mathsf{rec}[A](a; b; \beta)(M) : A}\, I\mathsf{rec}$$

where the β_0-rules that should hold (i.e. β-rules for 0-cells) are

$$\mathsf{rec}[A](a; b; \beta)(0_I) \equiv a$$
$$\mathsf{rec}[A](a; b; \beta)(1_I) \equiv b$$

and the β_1-rule is

$$(\mathsf{ap}_{\mathsf{rec}[A](a;b;\beta)}(\mathsf{seg}) =_{a=_A b} \beta)\ \mathsf{true}$$

That is to say, that the propositional equality above is inhabited.

From here, we can define the induction principle, which is similar to the recursor:

$$\frac{\Gamma, z : I \vdash A(z) : \mathcal{U} \quad \Gamma \vdash M : I \quad \Gamma \vdash a_0 : A(0_I) \quad \Gamma \vdash a_1 : A(1_I) \quad \Gamma \vdash p : a_0 =^{z.A}_{\mathsf{seg}} a_1}{\Gamma \vdash \mathsf{ind}[z.A](a_0; a_1; p)(M) : A(M)}\, I\mathsf{ind}$$

Recall that $a_0 =^{z.A}_{\mathsf{seg}} a_1$ is defined as $\mathsf{tr}[z.A](\mathsf{seg})(a_0) =_{A(1_I)} a_1$. This is because a_0 and a_1 no longer necessarily have the same type, so we need to use "path over" to express the desired equational law.

The β_0-rules for induction are

$$\mathsf{ind}[z.A](a_0; a_1; p)(0_I) \equiv a_0$$
$$\mathsf{ind}[z.A](a_0; a_1; p)(1_I) \equiv a_1$$

and the β_1-rule is

$$(\mathsf{dap}_{\mathsf{ind}[z.A](a_0;a_1;p)}(\mathsf{seg}) =_{a_0 =^{z.A}_{\mathsf{seg}} a_1} p)\ \mathsf{true}$$

There is also a unicity rule, or η-rule, that states "if a function behaves like the recursor of I, then it must be the recursor." There is a similar rule for the induction principle of I.

Exercise. *Define the η-rules for $I\mathsf{rec}$ and $I\mathsf{ind}$.*

3 The Circle Type $\mathbb{S}^1$

Another example of a higher inductive type is the circle type $\mathbb{S}^1$. The $\mathbb{S}^1$ type is defined by the 0-cell (point)

$$\mathsf{base} : \mathbb{S}^1$$

and the 1-cell (path)

$$\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$$

The recursor is defined as

$$\frac{\Gamma \vdash M : \mathbb{S}^1 \quad \Gamma \vdash a_0 : A \quad \Gamma \vdash l : a_0 =_A a_0}{\Gamma \vdash \mathsf{rec}[A](a_0; l)(M) : A}\ \mathbb{S}^1\mathsf{rec}$$

with the β_0-rule is

$$\mathsf{rec}[A](a_0; l)(\mathsf{base}) \equiv a_0$$

and the β_1-rule is

$$(\mathsf{ap}_{\mathsf{rec}[A](a_0;l)}(\mathsf{loop}) =_{a_0 =_A a_0} l)\ \mathsf{true}$$

The induction principle is defined as

$$\frac{\Gamma, z : \mathbb{S}^1 \vdash P(z) : \mathcal{U} \quad \Gamma \vdash M : \mathbb{S}^1 \quad \Gamma \vdash b : P(\mathsf{base}) \quad \Gamma \vdash l : b =^{z.P}_{\mathsf{loop}} b}{\Gamma \vdash \mathsf{ind}[z.P](b; l)(M) : P(M)}\ \mathbb{S}^1\mathsf{ind}$$

with β_0-rule

$$\mathsf{ind}[z.P](b; l)(\mathsf{base}) \equiv b$$

and β_1-rule

$$(\mathsf{dap}_{\mathsf{ind}[z.P](b;l)}(\mathsf{loop}) =_{b =^{z.P}_{\mathsf{loop}} b} l)\ \mathsf{true}$$

Be careful with the type of l, because it is easy to write a type that "typechecks" but is not correct. l should express that taking b around the loop path over P returns to b.

Exercise. *Define the η-rules for $\mathbb{S}^1\mathsf{rec}$ and $\mathbb{S}^1\mathsf{ind}$.*

4 Total Space of Loops as a Function from $\mathbb{S}^1$

Recall that we previously characterized the total space of paths,

$$\int \mathsf{Id}_A :\equiv \sum_{x,y:A} x =_A y$$

as being equivalent to the function type from I to A, i.e.

$$(I \to A) \simeq \int \mathsf{Id}_A$$

We can similarly characterize the total space of loops

$$\int \Omega_A :\equiv \sum_{x:A} x =_A x$$

as being equivalent to the function type from $\mathbb{S}^1$ to A. i.e.

$$(\mathbb{S}^1 \to A) \simeq \int \Omega_A$$

Proof. Define $f : (\mathbb{S}^1 \to A) \to \int \Omega_A$ as

$$f = \lambda g.\langle g(\mathsf{base}), \mathsf{ap}_g(\mathsf{loop})\rangle$$

Exercise. *Show that f has a quasi-inverse.*

□

5 Suspensions

Another example of a higher inductive type is the *suspension type*, which subsumes the interval and circle types (up to homotopy). For a type $A : \mathcal{U}$, the suspension of A, denoted by $\mathsf{susp}(A) : \mathcal{U}$, is the higher inductive type given by two 0-cell constructors

$$N : \mathsf{susp}(A)$$
$$S : \mathsf{susp}(A)$$

which will be referred to as the *north* and *south* poles, respectively, and a family of 1-cell constructors

$$\mathsf{mer} : A \to (N =_{\mathsf{susp}(A)} S)$$

which can be understood as a collection of *meridians*, i.e., paths from the north to the south pole.

Based on the above data, we can deduce the appropriate recursion schema: given any other type $B : \mathcal{U}$ which "looks like the suspension of A", there should be a function, called the recursor, from $\mathsf{susp}(A)$ to B, which preserves all the constructors. Expressing this formally, we have the following recursion rule

$$\frac{\Gamma \vdash B : \mathcal{U} \qquad \Gamma \vdash M : \mathsf{susp}(A) \qquad \Gamma \vdash b_N : B \qquad \Gamma \vdash b_S : B \qquad \Gamma, x : A \vdash m(x) : b_N =_B b_S}{\Gamma \vdash \mathsf{rec}[B](b_N; b_S; x.m(x))(M) : B}(\mathsf{susp}(A)\mathsf{rec})$$

Furthermore, the recursor behaves according to the following computation rules:

$$\mathsf{rec}[B](b_N; b_S; x.m(x))(N) \equiv b_N : B$$
$$\mathsf{rec}[B](b_N; b_S; x.m(x))(S) \equiv b_S : B$$
$$\mathsf{ap}_{\mathsf{rec}[B](b_N;b_S;x.m(x))(-)}(\mathsf{mer}(a)) =_{b_S =_B b_N} m(a)$$

The first two computation rules can be considered as β-rules for the 0-cells and the last one a β-rule for the 1-cells. Since we do not care about the specific witness term for the propositional equality in the conclusion of the last rule, we simply omit the witness.

We have an analogous induction schema, where instead of simple types $B : \mathcal{U}$ we consider dependent types $E : \mathsf{susp}(A) \to \mathcal{U}$. The induction rule states that in order to

construct a dependent function mapping $z : \mathsf{susp}(A)$ to an element of $E(z)$, it suffices to give elements $e_N : E(N)$ and $e_S : E(S)$ such that for each $x : A$, e_N and e_S are associated over the path $\mathsf{mer}(x)$. Formally, this means we have the following induction rule:

$$\frac{\begin{array}{c}\Gamma, z : \mathsf{susp}(A) \vdash E(z) : \mathcal{U} \\ \Gamma \vdash M : \mathsf{susp}(A) \qquad \Gamma \vdash e_N : E(N) \qquad \Gamma \vdash e_S : E(S) \\ \Gamma \vdash, x : A \vdash m(x) : e_N =^{z.E(z)}_{\mathsf{mer}(x)} e_S\end{array}}{\Gamma \vdash \mathsf{ind}[z.E(z)](e_N; e_S; x.m(x))(M) : E(M)}$$

Likewise, we have the following computation rules:

$$\begin{aligned}\mathsf{ind}[z.E(z)](e_N; e_S; x.m(x))(N) &\equiv e_N : E(N) \\ \mathsf{ind}[z.E(z)](e_N; e_S; x.m(x))(S) &\equiv e_S : E(S) \\ \mathsf{dap}_{\mathsf{ind}[z.E(z)](e_N;e_S;x.m(x))(-)}(\mathsf{mer}(a)) &=_{e_N =^{z.E(z)}_{\mathsf{mer}(x)} e_S} m(a)\end{aligned}$$

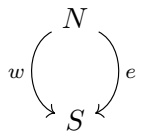
where the conclusion of the last rule refers to the application of a dependent function to a path, denoted by dap. We can state and prove a useful uniqueness principle, also known as the η-rule, asserting that "if a function behaves like the inductor, then it must be the inductor". We leave the exact statement of this principle and its proof as an exercise.

Why are we interested in suspensions in the first place? Interestingly, many familiar (and also not so familiar) inductive types can be characterized as suspensions. For example:

Exercise. *Show that the type* $\mathsf{susp}(0)$ *is equivalent to the type* $\mathbf{2}$.

Exercise. *Show that the type* $\mathsf{susp}(\mathbf{1})$ *is equivalent to the interval type* I.

What is $\mathsf{susp}(2)$? Since the type $\mathbf{2}$ contains only two elements (up to homotopy), we can picture $\mathsf{susp}(2)$ as a type generated by the two points N and S with two distinct paths between them, called w and e:



This of course looks very much like a circle - and indeed it is!

Exercise. *Show that the type* $\mathsf{susp}(\mathbf{2})$ *is equivalent to the circle type* $\mathbb{S}^1$.

Now we can ask the question, what is $\mathsf{susp}(\mathsf{susp}(2))$? The type $\mathsf{susp}(\mathsf{susp}(2))$ will of course have to contain the two points N and S. A function from $\mathsf{susp}(2)$ to the path space $N = S$ can then be thought of as a quadruple (w, e, γ, δ), where w,e are two distinct paths from N to S and γ, δ are two distinct paths from w to e. The type $\mathsf{susp}(\mathsf{susp}(2))$ generated by all this data can then be visualized as in Fig. 1:

This looks very much like a sphere - hence we can simply make the definition $\mathbb{S}^2 :\equiv \mathsf{susp}(\mathbb{S}^1)$. We can iterate this and set $\mathbb{S}^{n+1} :\equiv \mathsf{susp}(\mathbb{S}^n)$.

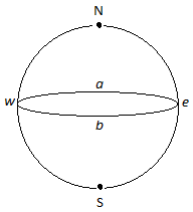


Figure 1: $\mathbb{S}^2$

15-819 Homotopy Type Theory
Lecture Notes

Matthew Maurer and Stefan Muller

January 1, 2014

1 Contents

2 Recap

Recall from last week the construction of the suspension of a type. Note that we are using different notation from the book, namely $\mathsf{Susp}(A)$ as opposed to $\sum A$. The suspension $\mathsf{Susp}(A)$ of A contains two 0-cells:

$$N : \mathsf{Susp}(A)$$

$$S : \mathsf{Susp}(A)$$

and paths $\mathsf{merid}(a)$ between N and S, where $a : A$.

$$\mathsf{merid} : \Pi x : A.N =_{\mathsf{Susp}(A)} S$$

$$x : \mathsf{Susp}(A) \vdash B(X) : \mathcal{U}$$

We also defined recursion and induction on $\mathsf{Susp}(A)$:

$$\frac{n : B \quad s : B \quad x : A \vdash m(x) : n =_B s}{z : \mathsf{Susp}(A) \vdash \mathsf{rec}[B](n; s; x.m) : B}$$

$$\frac{z : \mathsf{Susp}(A) \vdash B(z) : \mathcal{U} \quad n : B(N) \quad s : B(S) \quad x : A \vdash m(x) : n =^{z.B}_{\mathsf{merid}(x)} s}{z : \mathsf{Susp}(A) \vdash \mathsf{ind}[B](n; s; x.m) : B(z)}$$

3 Equivalence of $\mathsf{Susp}(2)$ and $\mathbb{S}^1$

Proposition 1.

$$\mathsf{Susp}(2) \simeq \mathbb{S}^1$$

Proof.

$$f : \mathsf{Susp}(2) \to \mathbb{S}^1$$

defined by

$$N \mapsto \mathsf{base}$$

$$S \mapsto \mathsf{base}$$

On the higher order part of the suspension, we define what the behavior of the one-cells should be, e.g. how ap would act.

$$\mathsf{merid}(tt) \mapsto \mathsf{loop}$$

$$\mathsf{merid}(ff) \mapsto \mathsf{refl}(\mathsf{base})$$

To show equivalence, we now define a quasinverse of f

$$g : \mathsf{Susp}(2) \to \mathbb{S}^1$$

where we send base is arbitrary but must be consistent

$$\mathsf{base} \mapsto N$$

$$\mathsf{loop} \mapsto \mathsf{merid}(tt) \cdot \mathsf{merid}(ff)^{-1}$$

Now need to prove these are inverses

$$\alpha : \Pi x : \mathsf{Susp}(2).g(f(x)) = x$$

By induction

$$\begin{array}{rl} (x = N) & \mathsf{refl}(N) : N = N \\ (x = S) & \mathsf{merid}(ff) : N = S \\ \mathsf{merid}(y) & ? : \mathsf{refl}(N) =^{z.f(g(z))=z}_{\mathsf{merid}(y)} \mathsf{merid}(ff) \end{array}$$

We can case out on this,

$$\mathsf{ap}_g(\mathsf{ap}_f(\mathsf{merid}(tt))^{-1} \cdot \mathsf{refl}(N) \cdot \mathsf{merid}(tt)) = \mathsf{merid}(ff)$$

$$\mathsf{ap}_g(\mathsf{ap}_f(\mathsf{merid}(ff))^{-1} \cdot \mathsf{refl}(N) \cdot \mathsf{merid}(ff)) = \mathsf{merid}(ff)$$

stepping evaluation,

$$\mathsf{ap}_g(\mathsf{loop})^{-1} \cdot \mathsf{merid}(tt) = \mathsf{merid}(ff)$$

$$(\mathsf{merid}(tt) \cdot \mathsf{merid}(ff)^{-1})^{-1} \cdot \mathsf{merid}(ff) = \mathsf{merid}(ff)^{-1-1} \cdot \mathsf{merid}(tt)^{-1} \cdot \mathsf{merid}(tt) = \mathsf{merid}(ff)$$

Our other proof of inversion

$$\beta : \Pi x : \mathbb{S}^1 . f(g(x)) = x$$

proceeds by induction on $\mathbb{S}^1$

$$\begin{array}{ll} (x = \mathsf{base}) & \mathsf{refl}(\mathsf{base}) : f(g(\mathsf{base})) = \mathsf{base} \\ (x = \mathsf{loop}) & \mathsf{ap}_f(\mathsf{ap}_g(\mathsf{loop}))^{-1} \cdot \mathsf{refl}(\mathsf{base}) = \mathsf{refl}(\mathsf{base}) \end{array}$$

□

4 Pointed Type

A pointed type is one with an example inhabitant. If A is a pointed type, then, in our notation, we have $a_0 : A$.

For example, $\Omega(A, a_0) = (a_0 =_A a_0)$, "the loop space", is pointed by $\mathsf{refl}(a_0)$. $\mathsf{Susp}(A)$ pointed by N (or S) is another example of a pointed type.

Pointed maps, also known as strict maps, are maps between two pointed types which map the well-known point of one to the well known point of the other, as in

$$(X, x_0) \multimap (Y, y_0) := \Sigma f : X \to Y . f(x_0) = y_0$$

We set out to prove

Proposition 2.

$$\mathsf{Susp}(A) \multimap B \simeq (A \multimap \Omega B)$$

Proof. Given $f : \mathsf{Susp}(A) \multimap B$, define $g : A \multimap \Omega B$ as

$$g(a) = p_0^{-1} \cdot \mathsf{ap}_f(\mathsf{merid}(a) \cdot \mathsf{merid}(a_0)^{-1}) \cdot p_0$$

where f_0 is the raw map, and p_0 is a proof of distinguished point preservation, e.g. $p_0 : f_0(N) = b_0$, and $f = \langle f_0, p_0 \rangle$

$$q = \mathsf{refl}(b_0)$$

On the other side, given $g : A \multimap \Omega B$, define $f : \mathsf{Susp}(A) \multimap B$

$$f_0(N) = b_0$$
$$f_0(S) = b_0$$

We again define the behavior of the one-cell in the HIT

$$\mathsf{ap}_f(\mathsf{merid}(a)) = g_0(a)$$

where g_0 is the raw map, and q_0 is a proof of distinguished point preservation, e.g. $q_0 : g_0(N) = b_0$ and $f = \langle g_0, q_0 \rangle$ □

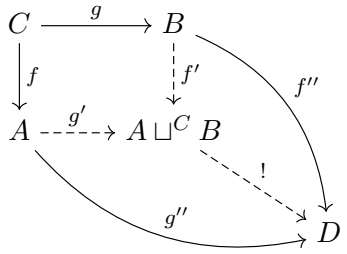
5 Pushouts

We temporarily return to conventional set math to define a pushout, which is dual to a pullback, which is an equationally constrained subset of a product. On the other hand, a pushout is essentially a disjoint union of two sets with some of the elements "glued" together. Specifically, assume we have sets A, B and C with maps f and g from C to A and B respectively. The pushout $A \sqcup^C B$ is the disjoint union of A and B with the images $f(C)$ and $g(C)$ merged by merging $f(c)$ and $g(c)$ together for all $c \in C$. In the notation of type theory, we denote the disjoint union of A and B as $A + B$ and use the maps $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$. We can then define the pushout as

$$A \sqcup^C B = (A + B)/R$$

where R is the least equivalence relation containing $\forall c \in C.R(\mathsf{inl}(f(c)), \mathsf{inr}(g(c)))$

$A \sqcup^C B$ is the "least such" object in the sense that it has a unique map to any other object D with the same properties:



where f' and g' are identified by f and g.

5.1 Pushouts of $A \xleftarrow{f} C \xrightarrow{g} B$

Moving back to HoTT, we can define pushouts as a higher inductive type, whose elements are those of the disjoint sum $A + B$, generated by mapping A and B to the pushout $A \sqcup^C B$ by inl and inr, respectively

$$\mathsf{inl} : A \to A \sqcup^C B$$
$$\mathsf{inr} : B \to A \sqcup^C B$$

and whose 1-cells connect $\mathsf{inl}(f(c))$ and $\mathsf{inr}(g(c))$ for every element c of C.

$$\mathsf{glue} : \Pi c : C.\mathsf{Id}_{A \sqcup^C B}(\mathsf{inl}(f(c)), \mathsf{inr}(g(c)))$$

As usual, we can define a recursor $\mathsf{rec}[D](\cdots) : A \sqcup^C B \to D$ (the map implied by the diagram above.) Note that for every element u of C, the recursor requires that there exist a path witnessing the equality of $[f(u)/x]l$ and $[g(u)/y]r$, representing the action of the recursor on glue and ensuring that the images of elements "glued together" in the pushout are also glued together in D.

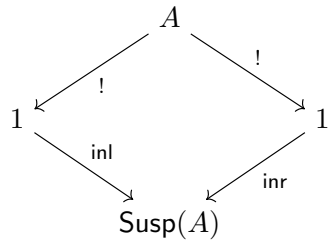
$$\frac{x : A \vdash l : D \quad y : B \vdash r : D \quad u : C \vdash q : [f(u)/x]l =_D [g(u)/y]r}{z : A \sqcup^C B \vdash \mathsf{rec}[D](x.l; y.r; u.q)(z) : D}$$

We have the usual β rules.

$$rec(x.l; y.r; u.q)(\mathsf{inl}(a)) \equiv [a/x]l$$
$$rec(x.l; y.r; u.q)(\mathsf{inr}(b)) \equiv [b/y]r$$
$$\mathsf{ap}_{rec(x.l;y.r;u.q)}(glue(c)) = [c/u]q$$

The idea of gluing together elements of a higher inductive type with a path for each element of a given type seems very reminiscent of suspensions. In fact, suspensions can be defined as pushouts $A \sqcup^C B$ with the types A and B and the maps f and g trivial:

$$\mathsf{Susp}(A) = 1 \sqcup^A 1$$



This immediately implies the following:

Corollary 1. *The pushout of two sets needn't be a set.*

This is true since $\mathbb{S}^1 = \mathsf{Susp}(2)$, but $\mathbb{S}^1$ is known not to be a set.

6 Quotients as HITs

The set definition of pushouts is in terms of a quotient, so we might consider defining quotients of sets by props uing HITs.

Consider the type-theoretic representation of A/R, a type A quotiented by the relation R. We define this type to have the normal properties of the quotient. We must be able to project an element of A into A/R, e.g. by computing its representative

$$q : A \to A/R$$

If we have two elements a and b of A that are related by R, that is, $R(a, b)$, then we must have a proof of equality

$$\mathsf{wd} : \Pi a, b : A.R(a, b) \to q(a) = q(b)$$

The above are the expected properties. However, we also wish for A/R to be a set. This requires that all proofs of equality in A/R must themselves be equal. We truncate A/R to a set by adding the required proofs of equality for all paths p, q between elements x and y.

$$\mathsf{trunc} : \Pi x, y : A/R.\Pi p, q : x =_{A/R} y.p = q$$

Saying there is a function from A/R to some type B is the same as saying there is a function from A to B that respects R by mapping related elements to elements that are (propositionally) equal in B.

Proposition 3.

$$(A/R) \to B \simeq \sum_{f:A\to B} \Pi_{a,b:A}.R(a,b) \to f(a) =_B f(b)$$

A proof appears in the textbook.

6.1 Example: $\mathbb{Z}$

We can represent integers as pairs of natural numbers (a, b) (representing the integer $a - b$). Since (infinitely) many pairs correspond to each integer, we quotient out by an appropriate relation.

$$\mathbb{Z} = (\mathbb{N} \times \mathbb{N})/R$$

where $R((a_1, b_1), (a_2, b_2))$ iff $a_1 + b_2 = a_2 + b_1$. This representation will be important in the proof that $\pi_1(\mathbb{S}^1) \simeq \mathbb{Z}$ in Section 8.

7 Truncations as HITs

We have previously seen the propositional truncation $||A||$, which truncates the type A to a proposition by forcing proofs of equality for all values. We now redefine the propositional truncation as a higher inductive type. Since we will generalize this to truncations for all h-levels n, we now write propositional truncation as $||A||_{-1}$.

The higher inductive type $||A||_{-1}$ has a simple constructor

$$|-|_{-1} : A \to ||A||_{-1}$$

To add the 1-cells, squash produces a proof of equality for any two values of A.

$$\mathsf{squash} : \Pi_{a,b:||A||_{-1}}.a =_{||A||_{-1}} b$$

The induction principle is as follows:

$$\frac{\begin{array}{c} z : ||A||_{-1} \vdash P(z) : \mathcal{U} \qquad x : A \vdash p : P(|x|_{-1}) \\ x, y : ||A||_{-1}, u : P(x), v : P(y) \vdash q : u =^{z.P}_{\mathsf{squash}(x,y)} v \end{array}}{z : ||A||_{-1} \vdash \mathsf{ind}[z.P](x.p; x, y, u, v.q)(z) : P(z)}$$

We can now use the same methodology to define set truncation as a HIT:

$$|-|_0 : A \to ||A||_0$$

As in the definition of quotients, the set truncation provides proofs of equality for all paths.

$$x, y : ||A||_0, p, q : x =_{||A||_0} y \vdash \mathsf{squash}(x, y, p, q) : p =_{x=y} q$$

And we state an induction principle:

$$\frac{\begin{array}{c} z : ||A||_0 \vdash P : U \qquad x : A \vdash g : P(|A|_0) \\ x, y : ||A||_0, z : P(x), w : P(y), p, q : x = y, r : z =_p^{z.P} w, s : z =_q^{z.P} w \vdash ip : p =_{\mathsf{squash}(x,y,p,q)}^{z.P} q \end{array}}{z : ||A||_0 \vdash \mathsf{ind}[z.P](x.g; x, y, z, w, p, q, r, s.ip)(z) : P(z)}$$

Proposition 4. *If A is a set, it is equivalent to its own set truncation, i.e.*

$$\mathsf{isSet}(A) \to ||A||_0 \simeq A$$

8 Fundamental group of $\mathbb{S}^1$

Recall $\mathbb{S}^1$ is a HIT defined by

$$\mathsf{base} : \mathbb{S}^1$$

$$\mathsf{loop} : \mathsf{base} =_{\mathbb{S}^1} \mathsf{base}$$

Recall $\Omega(A, a_0) := (a_0 =_A a_0)$
The fundamental group of A at a_0, $\pi_1(A, a_0)$, is defined to be $||\Omega(A, a_0)||_0$.

Theorem 1. *We want to show that*

$$\pi_1(\mathbb{S}^1) \simeq Z$$

Proof. Show $\Omega(\mathbb{S}^1, \mathsf{base}) \simeq \mathbb{Z}$ (this suffices by Proposition 4.)
We define a map

$$\mathsf{loop}^{(-)} : \mathbb{Z} \to \Omega(\mathbb{S}^1)$$

as follows:

$$\mathsf{loop}^{(0)} = \mathsf{refl}(\mathsf{base})$$

$$\mathsf{loop}^{(-n)} = \mathsf{loop}^{(-n+1)} \cdot \mathsf{loop}^{-1}$$

$$\mathsf{loop}^{(+n)} = \mathsf{loop}^{(n-1)} \cdot \mathsf{loop}$$

This could be, for example, defined by the $\mathbb{Z}$ recursor

$$winding : \Omega(\mathbb{S}^1) \to \mathbb{Z}$$

We take that $succ : \mathbb{Z} \simeq \mathbb{Z}$, and $pred = succ^{-1}$, defined by shifting all the numbers up/down by one

So, we have

$$ua(succ) : \mathbb{Z} =_{\mathcal{U}} \mathbb{Z}$$

By the recursion principle (not induction)

$$code : \mathbb{S}^1 \to \mathcal{U}$$

$$code(\mathsf{base}) = \mathbb{Z}$$

$$code(\mathsf{loop}) = ua(succ)$$

$$\mathsf{ap}_{code} : \Omega(\mathbb{S}^1) \to (\mathbb{Z} = \mathbb{Z})$$

$$winding(p) = tr[x.x](\mathsf{ap}_{code}(p))(0)$$

Proposition 5.

$$\Pi_{n:Z}.winding(\mathsf{loop}^{(n)}) =_Z n$$

Proof. Straightforward by induction □

Proposition 6.

$$\Pi_{l:\Omega(\mathbb{S}^1)}\mathsf{loop}^{winding(l)} =_{\Omega(\mathbb{S}^1)} l$$

We can't proceed by path induction on l, as the path is not free

$$encode : \Pi_{x:\mathbb{S}^1}(\mathsf{base} = x) \to code(x)$$

$$decode : \Pi_{x:\mathbb{S}^1} code(x) \to \mathsf{base} = x$$

$$encode(x, p) :\equiv tr[code](p)(o)$$

$$decode(x) :\equiv \mathsf{rec}_{\mathbb{S}^1}[z.code(z) \to \mathsf{base} = z](\lambda z.\mathsf{refl}(\mathsf{base}), \lambda n.\mathsf{loop}^{(n)})(x)$$

To complete the proof, you will need a path induction and a circle induction. □