Equality in Dependent Type Theory: Lecture Notes

Elias Abou Farah and Arman Ahmed

November 9, 2025

Contents

1 Motivation and Overview

In dependent type theory we distinguish two notions of equality: *judgmental (definitional) equality*, written $M \equiv N$, and *propositional equality*, a first-class *type* written $M =_A N$.[1] Judgmental equality is part of the meta-theory (convertibility via computation rules) and requires no inhabitant. Propositional equality is internalized as a type whose terms are *witnesses* that two elements of the same type are equal.

This separation lets us *state and prove* equalities that are not definitionally true (e.g. $x + 0 = x$ for symbolic x) while retaining a decidable type-checking core.

Judgmental equality (informal recap)

Judgmental equality identifies terms up to β-reduction, η (if present), and the ι-rules of inductive eliminators. Two key meta-rules used implicitly throughout are:
- **Conversion:** if $\Gamma \vdash M : A$ and $A \equiv B$, then $\Gamma \vdash M : B$.
- **Congruence (example for arrows):** if $A \equiv A'$ and (under $x{:}A$) $B \equiv B'$, then $(A \to B) \equiv (A' \to B')$.

2 Equality as an Identity Type

We introduce an *identity type* (propositional equality) with the usual three rules: formation, introduction (reflexivity), and elimination (the J-rule), together with its computation rule.

2.1 Formation

$$\frac{\Gamma \vdash A : \mathsf{Type} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash M =_A N : \mathsf{Type}}.$$

2.2 Introduction (Reflexivity)

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash \mathsf{refl}_A(M) : M =_A M}.$$

We often write simply refl when A and M are clear.

2.3 Elimination (J-rule) As Given in Lecture

We first note a (admittedly very large) J-rule as we have formulated all of our structures:

$$\frac{\Gamma \vdash P : N_1 =_A N_2 \quad \Gamma \vdash N_1 : A \quad \Gamma \vdash N_2 : A \quad \begin{array}{c} \Gamma,\ A : A \vdash M : [n/n_1,\ n/n_2,\ refl^A n/p]R \\ \Gamma,\ n_1 : A,\ n_1, p : n_1 =_A n_2 \vdash R : u \end{array}}{\begin{array}{c} \Gamma \vdash \mathsf{eqElim}\ P\ \mathsf{as}\ N_1 =_A N_2 \\ \mathsf{return}\ (n_1,\ n_2,\ p.R) \\ \mathsf{with}\ \mathsf{refl}^A n \Rightarrow M \end{array}}$$

At first look this is pretty incomprehensible, but, let's try to break this down:

- P is the proof of equality that we would like to eliminate, specifically it shows that $n_1 = n_2$.

---

[1] We follow standard presentations, e.g. [1, 2, 3].

- The type R is, in some sense, the proof we would like to spit out, though pre-substitution, so we should pick our R such that $[n/n_1,\ n/n_2,\ refl^A n/p]R$ is the type of our goal proof. We see that we pass it some variables that may prove useful in constructing this.

- We see that since our only way to introduce an equality is with refl, we can pattern-match on P with this assumption.

- Finally, M is the term of type R that proves our goal.

- The rest of the premises are just ensuring that everything is well-typed.

2.4 Elimination (J-rule / equality induction)

Let P be a family $P(x,y,p)$ of types depending on $x{:}A$, $y{:}A$, and $p{:}x =_A y$. If we can prove the *reflexive case* for all x,

$$d : \prod_{x:A} P(x,x,\mathsf{refl}(x)),$$

then for any $a,b{:}A$ and any $p{:}a =_A b$ we obtain

$$\mathsf{J}(d;\ a,b,p) : P(a,b,p)\,.$$

Computation rule.

$$\mathsf{J}(d;\ a,a,\mathsf{refl}(a)) \equiv d(a)\,,$$

i.e. eliminating a reflexivity proof computes to the corresponding branch.

2.5 Two derived tools: ap and transport

Lemma 2.1 (Action on paths). *Given $f{:}A \to B$ and $p{:}x =_A y$, there is $\mathsf{ap}(f,p){:}f(x) =_B f(y)$.*

Proof. By J on p with motive $P(x,y,p) \equiv f(x) =_B f(y)$; in the reflexive case we return $\mathsf{refl}(f(x))$. $\square$

Lemma 2.2 (Transport). *Let $P{:}A \to \mathsf{Type}$. For $p{:}x =_A y$ there is a map*

$$\mathsf{transport}^P(p) : P(x) \to P(y).$$

Proof. By J with motive $P'(x,y,p) \equiv (P(x) \to P(y))$; in the reflexive case use the identity function. $\square$

3 Worked Equalities

Lemma 3.1 (Right successor of +). *For all $n,m{:}\mathbb{N}$, $n + \mathsf{suc}(m) =_\mathbb{N} \mathsf{suc}(n + m)$.*

Proof. By induction on n.

Base. $n \equiv 0$. Then $0 + \mathsf{suc}(m) \equiv \mathsf{suc}(m) \equiv \mathsf{suc}(0 + m)$ by the defining equations for $+$, so take refl.

Step. Assume $q : n + \mathsf{suc}(m) =_\mathbb{N} \mathsf{suc}(n + m)$. Using the recursive clause for $+$ we obtain $(\mathsf{suc}n) + \mathsf{suc}(m) \equiv \mathsf{suc}(n + \mathsf{suc}(m))$. Apply $\mathsf{ap}(\mathsf{suc}, q)$ to rewrite the inner $n + \mathsf{suc}(m)$ to $\mathsf{suc}(n + m)$, yielding $(\mathsf{suc}n) + \mathsf{suc}(m) =_\mathbb{N} \mathsf{suc}(\mathsf{suc}(n + m))$ as required.

$\square$

3.1 Successor preserves equality

Theorem 3.2. *For all n, m:$\mathbb{N}$, if p:$n =_{\mathbb{N}} m$ then* $\mathsf{ap}(\mathsf{suc}, p) : \mathsf{suc}(n) =_{\mathbb{N}} \mathsf{suc}(m)$.

Proof. Immediate from Lemma 2.1 with $f \equiv \mathsf{suc}$. $\square$

3.2 Doubling equals adding a number to itself

Theorem 3.3. *For all n:$\mathbb{N}$,* $\mathsf{double}(n) =_{\mathbb{N}} n + n$.

Proof. By induction on n.

Base. $n \equiv 0$. Then $\mathsf{double}(0) \equiv 0 \equiv 0 + 0$ by computation; take refl.

Step. Assume $q : \mathsf{double}(n) =_{\mathbb{N}} n + n$. We must show $\mathsf{double}(\mathsf{suc}n) =_{\mathbb{N}} (\mathsf{suc}n) + (\mathsf{suc}n)$. Compute both sides:

$$\mathsf{double}(\mathsf{suc}n) \equiv \mathsf{suc}(\mathsf{suc}(\mathsf{double}(n))),$$
$$(\mathsf{suc}n) + (\mathsf{suc}n) \equiv \mathsf{suc}(n + \mathsf{suc}n) = \mathsf{suc}(\mathsf{suc}(n + n)) \quad \text{(by Lemma 3.1).}$$

Apply $\mathsf{ap}(\lambda t.\,\mathsf{suc}(\mathsf{suc}(t)), q)$ to rewrite $\mathsf{suc}(\mathsf{suc}(\mathsf{double}(n)))$ into $\mathsf{suc}(\mathsf{suc}(n + n))$, yielding the goal.

$\square$

3.3 A lecture example: if $x = \mathsf{suc}(0)$ then $\mathsf{double}(x) = \mathsf{suc}(\mathsf{suc}(0))$

Theorem 3.4. *For all x:$\mathbb{N}$, $x =_{\mathbb{N}} \mathsf{suc}(0) \to \mathsf{double}(x) =_{\mathbb{N}} \mathsf{suc}(\mathsf{suc}(0))$.*

Proof. Let p:$x =_{\mathbb{N}} \mathsf{suc}(0)$. Use J with motive

$$P(n_1, n_2, p) \equiv \mathsf{double}(n_1) =_{\mathbb{N}} \mathsf{double}(n_2).$$

The reflexive case returns refl for all n. Thus we obtain

$$\mathsf{double}(x) =_{\mathbb{N}} \mathsf{double}(\mathsf{suc}(0)).$$

By computation of double, $\mathsf{double}(\mathsf{suc}(0)) \equiv \mathsf{suc}(\mathsf{suc}(0))$. Conclude by conversion that $\mathsf{double}(x) =_{\mathbb{N}} \mathsf{suc}(\mathsf{suc}(0))$. $\square$

3.4 Successor is injective (fully detailed)

Theorem 3.5 (Successor injectivity). *For all n, m:$\mathbb{N}$, $\mathsf{suc}(n) =_{\mathbb{N}} \mathsf{suc}(m) \to n =_{\mathbb{N}} m$.*

Proof. Let p:$\mathsf{suc}(n) =_{\mathbb{N}} \mathsf{suc}(m)$. Apply Lemma 2.1 with $f \equiv \mathsf{pred}$ to obtain

$$\mathsf{ap}(\mathsf{pred}, p) : \mathsf{pred}(\mathsf{suc}(n)) =_{\mathbb{N}} \mathsf{pred}(\mathsf{suc}(m)).$$

By definition of pred, both endpoints compute to n and m respectively, so conversion finishes the proof. $\square$

3.5 Right identity of $+$ (expanded proof)

Theorem 3.6. *For all $n{:}\mathbb{N}$, $n + 0 =_{\mathbb{N}} n$.*

Proof. Proceed by induction on n.

Base. $n \equiv 0$. Then $0 + 0 \equiv 0$ by definition, so take refl.

Step. Assume $q : n + 0 =_{\mathbb{N}} n$. Using the definition of $+$ we compute $(\mathsf{suc}\, n) + 0 \equiv \mathsf{suc}(n + 0)$. Apply Theorem 3.2 with $p \equiv q$ to rewrite $\mathsf{suc}(n + 0)$ into $\mathsf{suc}(n)$, finishing the step.

$\square$

4 Intensional vs. Extensional Equality

In *intensional* Martin-Löf type theory, only the rules above are primitive; in particular there is no *equality reflection*

$$\frac{\Gamma \vdash p : M =_A N}{\Gamma \vdash M \equiv N : A},$$

and principles such as functional extensionality or univalence are not derivable without additional axioms [2, 3]. Adding equality reflection yields an *extensional* theory, but at the cost of complicating (and in general destroying) decidable type checking; standard proof assistants (Coq, Agda) adopt the intensional core.

5 Sigma Types and Existentials

The second lecture introduced the dependent pair (Σ) type, which internalizes *existential* statements. We keep the presentation close to [2].

5.1 Formation and judgmental equality

Given $A : \mathsf{Type}$ and a family $B : A \to \mathsf{Type}$, the dependent pair type

$$\Sigma_{x:A} B(x)$$

is itself a type. Its judgmental equality compares both components:

$$(\Sigma_{x:A} B(x)) \equiv (\Sigma_{x:A'} B'(x)) \quad \text{if } A \equiv A' \text{ and } B(x) \equiv B'(x) \text{ under } x{:}A.$$

The well-formedness side-conditions mirror those of Π-types: whenever $\Gamma \vdash A : \mathsf{Type}$ and $\Gamma, x{:}A \vdash B(x) : \mathsf{Type}$, we conclude $\Gamma \vdash \Sigma_{x:A} B(x) : \mathsf{Type}$.

5.2 Introduction and elimination

An inhabitant packages a witness and its dependent evidence:

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B(M)}{\Gamma \vdash \langle M, N \rangle : \Sigma_{x:A} B(x)}.$$

The dependent eliminator (pattern matching on pairs) has the usual form

$$\frac{\Gamma, x{:}A, y{:}B(x) \vdash C(x,y) : \mathsf{Type} \qquad \Gamma \vdash z : \Sigma_{x:A} B(x) \qquad \Gamma, x{:}A, y{:}B(x) \vdash d(x,y) : C(x,y)}{\Gamma \vdash \mathsf{split}(d, z) : C(\mathsf{fst}(z), \mathsf{snd}(z))},$$

with computation rule (the β-law) $\mathsf{split}(d, \langle M, N \rangle) \equiv d(M, N)$. Intuitively, the eliminator says that existential reasoning amounts to destructing a pair and continuing with a concrete witness and its certificate. The η-law complements this by expressing that any $z : \Sigma_{x:A} B(x)$ is judgmentally equal to the reassembled pair of its projections:

$$\langle \mathsf{fst}(z), \mathsf{snd}(z) \rangle \equiv z.$$

Together the β/η laws ensure that Σ behaves like the expected existential/product type up to definitional equality.

5.3   Derived projections and computation

The lecture often uses the non-dependent projections

$$\mathsf{fst} : \Sigma_{x:A} B(x) \to A, \qquad\qquad \mathsf{fst}(\langle M, N \rangle) \equiv M,$$
$$\mathsf{snd} : \prod_{z:\Sigma_{x:A} B(x)} B(\mathsf{fst}(z)), \qquad\qquad \mathsf{snd}(\langle M, N \rangle) \equiv N,$$

Or as premises and conclusions:

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash M : u \quad \Gamma \vdash N : [M/x]B \quad \Gamma \vdash \Sigma x : A.B}{\Gamma \vdash \mathsf{fst}\langle M,\ N \rangle = M : A}$$

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash M : u \quad \Gamma \vdash N : [M/x]B \quad \Gamma \vdash \Sigma x : A.B}{\Gamma \vdash \mathsf{snd}\langle M,\ N \rangle = N : [M/x]B}$$

which themselves are instances of split. The computation rules express that projecting the first component of a canonical pair returns the witness, and projecting the second component returns the evidence instantiated at that witness. These rules drive reasoning about Σ-types just as β/η rules do for functions.

Example 5.1 (Length-indexed sequences)**.** Let $\vec{(A}, n)$ be the usual inductive family of vectors of elements of A and length n. Then $\Sigma_{n:\mathbb{N}} \vec{(A}, n)$ packages a length together with the corresponding data, providing the "dependent pair" representation of dynamically sized vectors. The projections recover the runtime length $\mathsf{fst}(z) : \mathbb{N}$ and the payload $\mathsf{snd}(z) : \vec{(A}, \mathsf{fst}(z))$, while the η-law guarantees that rebuilding from these projections yields the original value. This is the standard way to swap between dependently typed interfaces and existential packages in systems such as Agda or Coq.

5.4   Existential specifications: the predecessor contract

With Σ available we can state existence properties cleanly. For example, the predecessor totality discussed in class is formulated as

$$\prod_{x:\mathbb{N}} \Big(x =_\mathbb{N} \mathbf{0} \to \mathbf{0} =_\mathbb{N} \mathsf{suc}(\mathbf{0}) \Big) \to \Sigma_{y:\mathbb{N}} \mathsf{suc}(y) =_\mathbb{N} x,$$

i.e. every non-zero natural has a predecessor whose successor is judgmentally equal to the original x. The proof follows the same inductive structure as our earlier recursor arguments: handle $x \equiv \mathbf{0}$ by contradiction, and in the successor case package the obvious witness $y \equiv n$ together with the reflexive equality proof. Erasing the proof-relevant pieces yields the expected computation for pred (this is the "program extraction" intuition highlighted in lecture).

5.5 Encoding negation via empty equalities

Although we have not introduced a primitive empty type, the class notes model falsity by the uninhabited equality $\mathbf{0} =_{\mathbb{N}} \mathsf{suc}(\mathbf{0}) \equiv \mathbf{0} =_{\mathbb{N}} \mathsf{suc}(\mathbf{0})$. Negation is then $A \to \mathbf{0} =_{\mathbb{N}} \mathsf{suc}(\mathbf{0}) \equiv A \to \mathbf{0} =_{\mathbb{N}} \mathsf{suc}(\mathbf{0})$. Because no term can inhabit $\mathbf{0} =_{\mathbb{N}} \mathsf{suc}(\mathbf{0})$, any function of type $A \to \mathbf{0} =_{\mathbb{N}} \mathsf{suc}(\mathbf{0})$ acts as a refutation of A. This trick suffices for small derivations (e.g. the "$x \neq 0$" premise above) until we extend the core with an explicit empty type.

6 Historical and Research Perspectives

6.1 Historical notes

Identity types enter the literature with Martin-Löf's 1970s formulations of constructive type theory, where equality witnesses replaced logical equivalence proofs [1]. The shift from external (judgmental) reasoning to internal identity proofs mirrors older distinctions in logic between definitional equality (dating back to Frege) and propositional equality (explicitly treated by Church). Modern expositions such as [2] highlight how J captures Leibniz's indiscernibility principle inside the theory, making the rules simultaneously computational and proof-theoretic.

6.2 Current research threads

Homotopy Type Theory (HoTT) reinterprets $M =_A N$ as a space of paths, leading to univalence and higher inductive types [3]. A major line of current work studies computational presentations of these ideas; examples include cubical type theory, where paths are functions out of an abstract interval object and univalence becomes a definitional equality [4], and Cartesian cubical frameworks that scale to higher-dimensional proof assistants and synthetic homotopy constructions [5]. These developments aim to reconcile rich equality principles with canonicity and computation, and motivate the transport and `ap` patterns emphasized in these notes.

7 Common Patterns and Tactics

- Prefer definitional simplification first (unfold and compute). Use propositional rewriting only for the remaining differences.

- Choose J's motive so that the reflexive branch becomes `refl` ("the branch carries no extra information").

- Use derived combinators: `ap` for function congruence; `transport` for dependent substitution.

8 Practice Problems

Practice 8.1 (Successor injectivity)**.** Prove: $\forall n, m{:}\mathbb{N}.\ \mathsf{suc}(n) =_{\mathbb{N}} \mathsf{suc}(m) \to n =_{\mathbb{N}} m$. *Hint:* Use J on the given equality with a motive that peels off one `suc` on both sides.

Practice 8.2 (Left and right identity of $+$)**.** (a) Prove by induction on x that $x + 0 = x$. (b) Prove that $0 + x = x$ (this one holds by computation with our definition).

Practice 8.3 (Successor on the right)**.** Reprove Lemma 3.1 (the right-successor law for $+$) directly by induction on n.

Practice 8.4 (Symmetry of equality)**.** Construct $\mathsf{sym} : \prod_{x,y:\mathbb{N}} x =_{\mathbb{N}} y \to y =_{\mathbb{N}} x$ using the J-rule. *Hint:* Choose the motive $P(x, y, p) \equiv y =_{\mathbb{N}} x$ so that the reflexive branch reduces to refl.

Practice 8.5 (Transport preserves evenness)**.** Define the predicate $\mathsf{Even} : \mathbb{N} \to \mathsf{Type}$ by $\mathsf{Even}(0) \equiv \top$ and $\mathsf{Even}(\mathsf{suc}(\mathsf{suc}n)) \equiv \mathsf{Even}(n)$ (with no constructor for odd inputs). Using Lemma 2.2, prove that any $p{:}n =_{\mathbb{N}} m$ induces a map $\mathsf{transport}^{\mathsf{Even}}(p) : \mathsf{Even}(n) \to \mathsf{Even}(m)$.

Practice 8.6 (A variant of Theorem 3.4)**.** Show: $\forall x{:}\mathbb{N}.\ x =_{\mathbb{N}} \mathsf{suc}(0) \to \mathsf{double}(x) =_{\mathbb{N}} 2$, where 2 is $\mathsf{suc}(\mathsf{suc}(0))$. Use the same motive as in Theorem 3.4.

Practice 8.7 (Sigma projections)**.** Define fst and snd using the Σ-eliminator and prove their computation rules $\mathsf{fst}(\langle M, N \rangle) \equiv M$ and $\mathsf{snd}(\langle M, N \rangle) \equiv N$.

Practice 8.8 (Predecessor as an existential)**.** Formalize the specification $\prod_{x:\mathbb{N}}(x =_{\mathbb{N}} \mathbf{0} \to \mathbf{0} =_{\mathbb{N}} \mathsf{suc}(\mathbf{0})) \to \Sigma_{y:\mathbb{N}}\mathsf{suc}(y) =_{\mathbb{N}} x$. Carry out the proof by recursion on x and explain where the "non-zero" hypothesis is used.

References

[1] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.

[2] Robert Harper. *Practical Foundations for Programming Languages.* Cambridge University Press, 2nd edition, 2016.

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

[4] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A constructive interpretation of the univalence axiom. In *Proc. TYPES*, 2018.

[5] Carlo Angiuli, Thierry Coquand, and Robert Harper. Cartesian cubical computational type theory: Constructive reasoning with paths and equalities. *Logical Methods in Computer Science*, 15(3), 2019.