The HoTT-Book

Chapters 1.4 through 1.7

HoTT-Seminar, SS2017

We'll discuss

1. Dependent function types (Π-types)
2. Product types
3. Dependent pair types (Σ-types)
4. Coproduct types

Dependent function types (Π-types)

Dependent function types (Π-types)

What are they? What do they want from us?

- A more general version of function types
- The elements of a Π-type are functions whose codomain type can vary depending on the element of the domain to which the function is applied, called dependent functions

Dependent function types (Π-types)

But can we construct it?

- Given a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, we may construct the type of dependent functions $\prod_{(x:A)} B(x) : \mathcal{U}$
- There are many alternative notations for this type, such as

$$\prod_{(x:A)} B(x) \qquad \prod_{(x:A)} B(x) \qquad \Pi(x : A),\ B(x).$$

- If B is a constant family, then the dependent product type is the ordinary function type:

$$\prod_{(x:A)} B \equiv (A \to B).$$

Dependent function types (Π-types)

May I introduce you?

- They are defined by explicit definitions
- To define $f : \prod_{(x:A)} B(x)$, where f is the name of a dependent function to be defined, we need an expression $\Phi : B(x)$ possibly involving the variable $x : A$
- So either like this:

$$f(x) :\equiv \Phi \qquad \text{for } x : A.$$

- Or alternatively using λ-abstraction:

$$\lambda x.\, \Phi \;:\; \prod_{x:A} B(x).$$

Dependent function types (Π-types)

Use the function, Luke.

- We can apply $f : \prod_{(x:A)} B(x)$ to an argument $a : A$ to obtain an element $f(a) : B(a)$
- The equalities are the same as for the ordinary function type, i.e. we have the computation rule given $a : A$
  - $f(a) \equiv \Phi[a/x]$
  - $(\lambda x.\, \Phi)(a) \equiv \Phi[a/x]$
- Similarly, we have the uniqueness principle

$$f \equiv (\lambda x.\, f(x)) \qquad \text{for any } f : \prod_{x:A} B(x)$$

Dependent function types (Π-types)

The example we deserve, not the one we need right now

- There is a type family $\mathrm{Fin} : \mathbb{N} \to \mathcal{U}$ whose values are the standard finite sets with elements $0_n, 1_n, \ldots, (n-1)_n : \mathrm{Fin}(n)$
- There is then a dependent function $\mathrm{fmax} : \prod_{(n:\mathbb{N})} \mathrm{Fin}(n+1)$ which returns the "largest" element of each nonempty finite type, $\mathrm{fmax}(n) :\equiv n_{n+1}$
- As was the case for Fin itself, we cannot define fmax yet, but we will be able to soon
- But not today
- Possibly not tommorow either

Dependent function types (Π-types)

Polymorphisms, man's best friend

- We can now define functions which are polymorphic over a given universe.
- A polymorphic function is one which takes a type as one of its arguments, and then acts on elements of that type (or of other types constructed from it)
- The most trivial example being the identity-function:

$$\mathrm{id} : \prod_{A:\mathcal{U}} A \to A$$

which we define by

$$\mathrm{id} :\equiv \lambda(A:\mathcal{U}).\, \lambda(x:A).\, x$$

or with an argument as a subscript

$$\mathrm{id}_A(x) :\equiv x$$

Dependent function types (Π-types)

Swapalaludubdub

- What does this function do?

$$\text{swap} : \prod_{(A:\mathcal{U})} \prod_{(B:\mathcal{U})} \prod_{(C:\mathcal{U})} (A \to B \to C) \to (B \to A \to C).$$

$$\text{swap}(A, B, C, g) :\equiv \lambda b.\, \lambda a.\, g(a)(b).$$

- It switches the order of the arguments of a (curried) two-argument function
- We could have also defined it like this:

$$\text{swap}_{A,B,C}(g)(b, a) :\equiv g(a, b).$$

- And typed it like this:

$$\text{swap} : \prod_{A,B,C:\mathcal{U}} (A \to B \to C) \to (B \to A \to C).$$

Dependent function types (Π-types)

To dependency and beyond!

- In this case the dependent swap function is curried:

$$\text{swap} : \prod_{(A:\mathcal{U})} \prod_{(B:\mathcal{U})} \prod_{(C:\mathcal{U})} (A \to B \to C) \to (B \to A \to C).$$

- However, in the dependent case the second domain may depend on the first one, and the codomain may depend on both
- Given $A : \mathcal{U}$ and type families $B : A \to \mathcal{U}$ and $C : \prod_{(x:A)} B(x) \to \mathcal{U}$, we may construct the type

$$g : \prod_{(x:A)} \prod_{(y:B(x))} C(x, y)$$

of functions with two arguments; and given $a : A$ and $b : B(a)$ we get the type $g(a, b) : C(a, b)$

Product types

Product types

Y'all got anymore of them product types?

- Given types $A, B : \mathcal{U}$ we introduce the type $A \times B : \mathcal{U}$, which we call their cartesian product
- We intend the elements of $A \times B$ to be pairs $(a, b) : A \times B$, where $a : A$ and $b : B$
- We also introduce a nullary product type, called the unit type $\mathbf{1} : \mathcal{U}$.
- And the only element of $\mathbf{1}$ to be some particular object $\star : \mathbf{1}$

Product types

Products, assemble!

- Given $a : A$ and $b : B$, we may form $(a, b) : A \times B$.
- Similarly, there is a unique way to construct elements of $\mathbf{1}$, namely we have $\star : \mathbf{1}$.
- We expect that "every element of $A \times B$ is a pair", which is the uniqueness principle for products; we do not assert this as a rule of type theory, but we will prove it later on as a propositional equality.
- In a couple of slides that is

Product types

I am your function

- How do we define functions using a product type?
- Let us first consider the definition of a non-dependent function $f : A \times B \to C$
- Since we intend the only elements of $A \times B$ to be pairs, we expect to be able to define such a function by prescribing the result when f is applied to a pair (a, b) We can prescribe these results by providing a function $g : A \to B \to C$..
- for any such g, we can define a function $f : A \times B \to C$ by

$$f((a, b)) :\equiv g(a)(b).$$

Product types

I love the smell of projections in the morning

- As an example, we can derive the projection functions:

$$\mathrm{pr}_1 : A \times B \to A$$
$$\mathrm{pr}_2 : A \times B \to B$$

  with the defining equations

$$\mathrm{pr}_1((a, b)) :\equiv a$$
$$\mathrm{pr}_2((a, b)) :\equiv b.$$

- They look rather similar, we could try to construct a universal function for product types and redefine our projections by applying that function to specific arguments.

Product types

Keep your functions close, but your recursors closer

- The recursor might accomplish that:

$$\mathrm{rec}_{A\times B} : \prod_{C:\mathcal{U}} (A \to B \to C) \to A \times B \to C$$

with the defining equation

$$\mathrm{rec}_{A\times B}(C, g, (a, b)) :\equiv g(a)(b).$$

- Then instead of defining functions such as pr_1 and pr_2 directly by a defining equation, we could define

$$\mathrm{pr}_1 :\equiv \mathrm{rec}_{A\times B}(A, \lambda a.\, \lambda b.\, a)$$
$$\mathrm{pr}_2 :\equiv \mathrm{rec}_{A\times B}(B, \lambda a.\, \lambda b.\, b).$$

Product types

You're gonna need a bigger type

- We also have a recursor for the unit type:

$$\mathrm{rec}_{\mathbf{1}} : \prod_{C:\mathcal{U}} C \to \mathbf{1} \to C$$

  with the defining equation

$$\mathrm{rec}_{\mathbf{1}}(C, c, \star) :\equiv c.$$

- [...] the recursor for **1** is completely useless [...]

Product types

Say hello to my dependent function

- To be able to define dependent functions over the product type, we have to generalize the recursor
- Given $C : A \times B \to \mathcal{U}$, we may define a function $f : \prod_{(x:A\times B)} C(x)$ by providing a function

$$g : \prod_{(x:A)} \prod_{(y:B)} C((x, y))$$

with defining equation

$$f((x, y)) :\equiv g(x)(y).$$

Product types

Proofs? Where we're going we don't need proofs. (Well actually we do)

- For example, in this way we can prove the propositional uniqueness principle, which says that every element of $A \times B$ is equal to a pair
- Specifically, we can construct a function

$$\mathsf{uniq}_{A\times B} : \prod_{x:A\times B} ((\mathsf{pr}_1(x), \mathsf{pr}_2(x)) =_{A\times B} x).$$

- There is a reflexivity element $\mathsf{refl}_x : x =_A x$ for any $x : A$.
- Given this, we can define

$$\mathsf{uniq}_{A\times B}((a, b)) :\equiv \mathsf{refl}_{(a,b)}.$$

In the case that $x :\equiv (a, b)$ we can calculate

$$(\mathsf{pr}_1((a, b)), \mathsf{pr}_2((a, b))) \equiv (a, b)$$

using the defining equations for the projections. Therefore,

$$\mathsf{refl}_{(a,b)} : (\mathsf{pr}_1((a, b)), \mathsf{pr}_2((a, b))) = (a, b)$$

Product types

Hello induction my old friend

- To prove a property for all elements of a product, it is enough to prove it for its canonical elements, the ordered pairs
- Induction for product types: given $A, B : \mathcal{U}$ we have

$$\mathrm{ind}_{A\times B} : \prod_{C:A\times B\to\mathcal{U}} \left(\prod_{(x:A)} \prod_{(y:B)} C((x,y)) \right) \to \prod_{x:A\times B} C(x)$$

with the defining equation

$$\mathrm{ind}_{A\times B}(C, g, (a,b)) :\equiv g(a)(b).$$

- The recursor for comparison:

$$\mathrm{rec}_{A\times B} : \prod_{C:\mathcal{U}} (A \to B \to C) \to A \times B \to C$$

- Induction for the unit type turns out to be more useful than the recursor:

$$\mathsf{ind}_{\mathbf{1}} : \prod_{C:\mathbf{1}\to\mathcal{U}} C(\star) \to \prod_{x:\mathbf{1}} C(x)$$

with the defining equation

$$\mathsf{ind}_{\mathbf{1}}(C, c, \star) :\equiv c.$$

- Induction enables us to prove the propositional uniqueness principle for $\mathbf{1}$, which asserts that its only inhabitant is $\star$

$$\mathsf{uniq}_{\mathbf{1}} : \prod_{x:\mathbf{1}} x = \star$$

$$\mathsf{uniq}_{\mathbf{1}} :\equiv \mathsf{ind}_{\mathbf{1}}(\lambda x.\, x = \star, \mathsf{refl}_{\star}).$$

Dependent pair types (Σ-types)

Dependent pair types (Σ-types)

You had me at dependent

- These are a more general version of product types
- We can allow the type of the second component of a pair to vary depending on the choice of the first

Dependent pair types (Σ-types)

We'll always have pairs

- Given a type $A : \mathcal{U}$ and a family $B : A \to \mathcal{U}$, the dependent pair type is written as $\sum_{(x:A)} B(x) : \mathcal{U}$.
- Alternative notations are

$$\sum_{(x:A)} B(x) \qquad \sum_{(x:A)} B(x) \qquad \sum(x : A),\ B(x).$$

- If B is constant, then the dependent pair type is the ordinary cartesian product type:

$$\left(\sum_{x:A} B\right) \equiv (A \times B).$$

Dependent pair types (Σ-types)

You must construct additional pair types

- The way to construct elements of a dependent pair type is by pairing: we have $(a, b) : \sum_{(x:A)} B(x)$ given $a : A$ and $b : B(a)$.
- All the constructions on Σ-types arise as straightforward generalizations of the ones for product types, with dependent functions often replacing non-dependent ones.

Dependent pair types (Σ-types)

Shut up and take my recursor

- For instance, the recursion principle says that to define a non-dependent function out of a Σ-type $f : \left(\sum_{(x:A)} B(x)\right) \to C$, we provide a function $g : \prod_{(x:A)} B(x) \to C$, and then we can define f via the defining equation

$$f((a, b)) :\equiv g(a)(b).$$

- For instance, we can derive the first projection from a Σ-type

$$\mathrm{pr}_1 : \left(\sum_{x:A} B(x)\right) \to A.$$

by the defining equation

$$\mathrm{pr}_1((a, b)) :\equiv a.$$

Dependent pair types (Σ-types)

Something something second projection

- However, since the type of the second component of a pair

$$(a, b) : \sum_{x:A} B(x)$$

is $B(a)$, the second projection must be a *dependent* function, whose type involves the first projection function:

$$\mathrm{pr}_2 : \prod_{p:\sum_{(x:A)} B(x)} B(\mathrm{pr}_1(p)).$$

Dependent pair types (Σ-types)

We use the induction principle. It's super effective

- Thus we need the *induction* principle for Σ-types. This says that to construct a dependent function out of a Σ-type into a family $C : (\sum_{(x:A)} B(x)) \to \mathcal{U}$, we need a function

$$g : \prod_{(a:A)} \prod_{(b:B(a))} C((a, b)).$$

- We can then derive a function

$$f : \prod_{p:\sum_{(x:A)} B(x)} C(p)$$

with defining equation

$$f((a, b)) :\equiv g(a)(b).$$

Dependent pair types (Σ-types)

At this point I ran out of ideas, this is still about the second projection

- Applying this with $C(p) :\equiv B(\mathrm{pr}_1(p))$, we can define

$$\mathrm{pr}_2 : \prod_{p:\sum_{(x:A)} B(x)} B(\mathrm{pr}_1(p))$$

  with the obvious equation

$$\mathrm{pr}_2((a,b)) :\equiv b.$$

- To convince ourselves that this is correct, we note that $B(\mathrm{pr}_1((a,b))) \equiv B(a)$, using the defining equation for pr_1, and indeed $b : B(a)$.

Dependent pair types (Σ-types)

Recursor and induction, again...

- We can package the recursion and induction principles into the recursor for Σ:

$$\mathrm{rec}_{\sum_{(x:A)} B(x)} : \prod_{(C:\mathcal{U})} \left(\prod_{(x:A)} B(x) \to C\right) \to \left(\sum_{(x:A)} B(x)\right) \to C$$

with the defining equation

$$\mathrm{rec}_{\sum_{(x:A)} B(x)}(C, g, (a, b)) :\equiv g(a)(b)$$

- and the corresponding induction operator:

$$\mathrm{ind}_{\sum_{(x:A)} B(x)} : \prod_{(C:(\sum_{(x:A)} B(x))\to\mathcal{U})} \left(\prod_{(a:A)} \prod_{(b:B(a))} C((a, b))\right) \to \prod_{(p:\sum_{(x:A)} B(x))} C(p)$$

with the defining equation

$$\mathrm{ind}_{\sum_{(x:A)} B(x)}(C, g, (a, b)) :\equiv g(a)(b).$$

Coproduct types

Coproduct types

It just works, and here's how

- Given $A, B : \mathcal{U}$, we introduce their **coproduct** type $A + B : \mathcal{U}$.
- This corresponds to the *disjoint union* in set theory, and we may also use that name for it
- In type theory, as was the case with functions and products, the coproduct must be a fundamental construction, since there is no previously given notion of "union of types"
- We also introduce a nullary version: the **empty type 0** $: \mathcal{U}$.

Coproduct types

Construction. It's that simple

- There are two ways to construct elements of $A + B$:
- either as $\mathrm{inl}(a) : A + B$ for $a : A$, or as $\mathrm{inr}(b) : A + B$ for $b : B$.
- There are no ways to construct elements of the empty type

Coproduct types

Oh god not the recursor again

- To construct a non-dependent function $f : A + B \to C$, we need functions $g_0 : A \to C$ and $g_1 : B \to C$
- Then f is defined via the defining equations

$$f(\mathsf{inl}(a)) :\equiv g_0(a),$$
$$f(\mathsf{inr}(b)) :\equiv g_1(b).$$

- That is, the function f is defined by **case analysis**. As before, we can derive the recursor:

$$\mathsf{rec}_{A+B} : \prod_{(C:\mathcal{U})} (A \to C) \to (B \to C) \to A + B \to C$$

with the defining equations

$$\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inl}(a)) :\equiv g_0(a),$$
$$\mathsf{rec}_{A+B}(C, g_0, g_1, \mathsf{inr}(b)) :\equiv g_1(b).$$

Coproduct types

And look, even the empty type has one

- We can always construct a function $f : \mathbf{0} \to C$ without having to give any defining equations, because there are no elements of $\mathbf{0}$ on which to define f.
- Thus, the recursor for $\mathbf{0}$ is

$$\mathrm{rec}_{\mathbf{0}} : \prod_{(C:\mathcal{U})} \mathbf{0} \to C,$$

  which constructs the canonical function from the empty type to any other type.
- Logically, it corresponds to the principle *ex falso quodlibet*

Coproduct types

You know where this is going...

- To construct a dependent function $f : \prod_{(x:A+B)} C(x)$ out of a coproduct, we assume as given the family $C : (A + B) \to \mathcal{U}$, and require

$$g_0 : \prod_{a:A} C(\mathsf{inl}(a)),$$
$$g_1 : \prod_{b:B} C(\mathsf{inr}(b)).$$

- This yields f with the defining equations:

$$f(\mathsf{inl}(a)) :\equiv g_0(a),$$
$$f(\mathsf{inr}(b)) :\equiv g_1(b).$$

Coproduct types

... yup, induction. Who would've guessed

- We package this scheme into the induction principle for coproducts:

$$\mathrm{ind}_{A+B} : \prod_{(C:(A+B)\to\mathcal{U})} \left(\prod_{(a:A)} C(\mathrm{inl}(a))\right) \to \left(\prod_{(b:B)} C(\mathrm{inr}(b))\right) \to \prod_{(x:A+B)} C(x).$$

- The induction principle for the empty type

$$\mathrm{ind}_{\mathbf{0}} : \prod_{(C:\mathbf{0}\to\mathcal{U})} \prod_{(z:\mathbf{0})} C(z)$$

gives us a way to define a trivial dependent function out of the empty type.

Appendix A

Dependent function types ($\prod$-types)

- We introduce a primitive constant $c_{\prod}$, but write $c_{\prod}(A, \lambda x.\, B)$ as $\prod_{(x:A)} B$. Judgments concerning such expressions and expressions of the form $\lambda x.\, b$ are introduced by the following rules:
- if $\Gamma \vdash A : \mathcal{U}_n$ and $\Gamma, x : A \vdash B : \mathcal{U}_n$, then $\Gamma \vdash \prod_{(x:A)} B : \mathcal{U}_n$
- if $\Gamma, x : A \vdash b : B$ then $\Gamma \vdash (\lambda x.\, b) : (\prod_{(x:A)} B)$
- if $\Gamma \vdash g : \prod_{(x:A)} B$ and $\Gamma \vdash t : A$ then $\Gamma \vdash g(t) : B[t/x]$
- If x does not occur freely in B, we abbreviate $\prod_{(x:A)} B$ as the non-dependent function type $A \to B$ and derive the following rule: if $\Gamma \vdash g : A \to B$ and $\Gamma \vdash t : A$ then $\Gamma \vdash g(t) : B$

Dependent function types (Π-types)

- Using non-dependent function types and leaving implicit the context Γ, the rules above can be written in the following alternative style that we use in the rest of this section of the appendix:
- if $A : \mathcal{U}_n$ and $B : A \to \mathcal{U}_n$, then $\prod_{(x:A)} B(x) : \mathcal{U}_n$
- if $x : A \vdash b : B$ then $\lambda x.\, b : \prod_{(x:A)} B(x)$
- if $g : \prod_{(x:A)} B(x)$ and $t : A$ then $g(t) : B(t)$

Dependent pair types (Σ-types)

- We introduce primitive constants c_Σ and c_{pair}. An expression of the form $c_\Sigma(A, \lambda a.\, B)$ is written as $\sum_{(a:A)} B$, and an expression of the form $c_{\mathrm{pair}}(a, b)$ is written as (a, b). We write $A \times B$ instead of $\sum_{(x:A)} B$ if x is not free in B.
- Judgments concerning such expressions are introduced by the following rules:
- if $A : \mathcal{U}_n$ and $B : A \to \mathcal{U}_n$, then $\sum_{(x:A)} B(x) : \mathcal{U}_n$
- if, in addition, $a : A$ and $b : B(a)$, then $(a, b) : \sum_{(x:A)} B(x)$

Dependent pair types (Σ-types)

- If we have A and B as above, $C : \left(\sum_{(x:A)} B(x)\right) \to \mathcal{U}_m$, and

$$d : \prod_{(x:A)} \prod_{(y:B(x))} C((x, y))$$

we can introduce a defined constant

$$f : \prod_{(p:\sum_{(x:A)} B(x))} C(p)$$

with the defining equation

$$f((x, y)) :\equiv d(x, y).$$

- Note that C, d, x, and y may contain extra implicit parameters $x_1, \ldots, x_n$ if they were obtained in some non-empty context; therefore, the fully explicit recursion schema is
$f(x_1, \ldots, x_n, (x(x_1, \ldots, x_n), y(x_1, \ldots, x_n))) :\equiv$
$d(x_1, \ldots, x_n, (x(x_1, \ldots, x_n), y(x_1, \ldots, x_n)))$

Coproduct types

- We introduce primitive constants c_+, c_{inl}, and c_{inr}. We write $A + B$ instead of $c_+(A, B)$, $\mathsf{inl}(a)$ instead of $c_{\mathsf{inl}}(a)$, and $\mathsf{inr}(a)$ instead of $c_{\mathsf{inr}}(a)$:
- if $A, B : \mathcal{U}_n$ then $A + B : \mathcal{U}_n$
- moreover, $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$
- If we have A and B as above, $C : A + B \to \mathcal{U}_m$, $d : \prod_{(x:A)} C(\mathsf{inl}(x))$, and $e : \prod_{(y:B)} C(\mathsf{inr}(y))$, then we can introduce a defined constant $f : \prod_{(z:A+B)} C(z)$ with the defining equations
- $$f(\mathsf{inl}(x)) :\equiv d(x) \qquad \text{and} \qquad f(\mathsf{inr}(y)) :\equiv e(y).$$

The finite types

- We introduce primitive constants $\star$, $\mathbf{0}$, $\mathbf{1}$, satisfying the following rules:
- $\mathbf{0} : \mathcal{U}_0$, $\mathbf{1} : \mathcal{U}_0$
- $\star : \mathbf{1}$
- Given $C : \mathbf{0} \to \mathcal{U}_n$ we can introduce a defined constant $f : \prod_{(x:\mathbf{0})} C(x)$, with no defining equations.
  Given $C : \mathbf{1} \to \mathcal{U}_n$ and $d : C(\star)$ we can introduce a defined constant $f : \prod_{(x:\mathbf{1})} C(x)$, with defining

Rules for type formers

- a **formation rule**, stating when the type former can be applied;
- some **introduction rules**, stating how to inhabit the type;
- **elimination rules**, or an induction principle, stating how to use an element of the type;
- **computation rules**, which are judgmental equalities explaining what happens when elimination rules are applied to results of introduction rules;
- optional **uniqueness principles**, which are judgmental equalities explaining how every element of the type is uniquely determined by the results of elimination rules applied to it.

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash B : \mathcal{U}_i}{\Gamma \vdash \prod_{(x:A)} B : \mathcal{U}_i}\ \Pi\text{-FORM}$$

$$\frac{\Gamma, x{:}A \vdash b : B}{\Gamma \vdash \lambda(x:A).\, b : \prod_{(x:A)} B}\ \Pi\text{-INTRO}$$

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a/x]}\ \Pi\text{-ELIM}$$

$$\frac{\Gamma, x{:}A \vdash b : B \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x:A).\, b)(a) \equiv b[a/x] : B[a/x]}\ \Pi\text{-COMP}$$

$$\frac{\Gamma \vdash f : \prod_{(x:A)} B}{\Gamma \vdash f \equiv (\lambda x.\, f(x)) : \prod_{(x:A)} B}\ \Pi\text{-UNIQ}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma, x{:}A \vdash B : \mathcal{U}_i}{\Gamma \vdash \sum_{(x:A)} B : \mathcal{U}_i}\ \Sigma\text{-FORM}$$

$$\frac{\Gamma, x{:}A \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \sum_{(x:A)} B}\ \Sigma\text{-INTRO}$$

$$\frac{\begin{array}{c}\Gamma, z{:}\sum_{(x:A)} B \vdash C : \mathcal{U}_i \\ \Gamma, x{:}A, y{:}B \vdash g : C[(x, y)/z] \qquad \Gamma \vdash p : \sum_{(x:A)} B\end{array}}{\Gamma \vdash \mathrm{ind}_{\sum_{(x:A)} B}(z.C, x.y.g, p) : C[p/z]}\ \Sigma\text{-ELIM}$$

$$\frac{\begin{array}{c}\Gamma, z{:}\sum_{(x:A)} B \vdash C : \mathcal{U}_i \qquad \Gamma, x{:}A, y{:}B \vdash g : C[(x, y)/z] \\ \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]\end{array}}{\Gamma \vdash \mathrm{ind}_{\sum_{(x:A)} B}(z.C, x.y.g, (a, b)) \equiv g[a, b/x, y] : C[(a, b)/z]}\ \Sigma\text{-COMP}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i}{\Gamma \vdash A + B : \mathcal{U}_i} \; +\text{-FORM}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}(a) : A + B} \; +\text{-INTRO}_1$$

$$\frac{\Gamma \vdash A : \mathcal{U}_i \qquad \Gamma \vdash B : \mathcal{U}_i \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}(b) : A + B} \; +\text{-INTRO}_2$$

$$\frac{\begin{array}{c}\Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \\ \Gamma, x{:}A \vdash c : C[\mathsf{inl}(x)/z] \qquad \Gamma, y{:}B \vdash d : C[\mathsf{inr}(y)/z] \\ \Gamma \vdash e : A+B\end{array}}{\Gamma \vdash \mathsf{ind}_{A+B}(z.C, x.c, y.d, e) : C[e/z]}\ +\text{-ELIM}$$

$$\frac{\begin{array}{c}\Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \\ \Gamma, x{:}A \vdash c : C[\mathsf{inl}(x)/z] \qquad \Gamma, y{:}B \vdash d : C[\mathsf{inr}(y)/z] \\ \Gamma \vdash a : A\end{array}}{\Gamma \vdash \mathsf{ind}_{A+B}(z.C, x.c, y.d, \mathsf{inl}(a)) \equiv c[a/x] : C[\mathsf{inl}(a)/z]}\ +\text{-COMP}_1$$

$$\frac{\begin{array}{c}\Gamma, z{:}(A+B) \vdash C : \mathcal{U}_i \\ \Gamma, x{:}A \vdash c : C[\mathsf{inl}(x)/z] \qquad \Gamma, y{:}B \vdash d : C[\mathsf{inr}(y)/z] \\ \Gamma \vdash b : B\end{array}}{\Gamma \vdash \mathsf{ind}_{A+B}(z.C, x.c, y.d, \mathsf{inr}(b)) \equiv d[b/y] : C[\mathsf{inr}(b)/z]}\ +\text{-COMP}_2$$

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbf{0} : \mathcal{U}_i}\ \mathbf{0}\text{-FORM} \qquad \frac{\Gamma, x{:}\mathbf{0} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash a : \mathbf{0}}{\Gamma \vdash \mathsf{ind}_{\mathbf{0}}(x.C, a) : C[a/x]}\ \mathbf{0}\text{-ELIM}$$

$$\frac{\Gamma \text{ ctx}}{\Gamma \vdash \mathbf{1} : \mathcal{U}_i}\ \mathbf{1}\text{-FORM} \qquad \frac{\Gamma \text{ ctx}}{\Gamma \vdash \star : \mathbf{1}}\ \mathbf{1}\text{-INTRO}$$

$$\frac{\Gamma, x{:}\mathbf{1} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\star/x] \qquad \Gamma \vdash a : \mathbf{1}}{\Gamma \vdash \mathsf{ind}_{\mathbf{1}}(x.C, c, a) : C[a/x]}\ \mathbf{1}\text{-ELIM}$$

$$\frac{\Gamma, x{:}\mathbf{1} \vdash C : \mathcal{U}_i \qquad \Gamma \vdash c : C[\star/x]}{\Gamma \vdash \mathsf{ind}_{\mathbf{1}}(x.C, c, \star) \equiv c : C[\star/x]}\ \mathbf{1}\text{-COMP}$$

The End

Questions?