Homotopy Type Theory Seminar Notes (Session 2)

Franklin

July 21, 2022

1 Product types

Last time, we learned about *inductive types*, which are defined in terms of their *constructors*, or all possible ways of producing elements of the type, and their *recursion principles*, which provide a way of producing functions out of the type. (Later on, we will also see something called an *induction principle* for each inductive type.) During this session, we learned about two ways of "combining" preexisting types A and B to produce new types $A \times B$ and $A + B$.

Given types A and B, we may construct a **product type** denoted $A \times B$. This is an inductive type with a single constructor called `pair`, which has the following type signature:

$$\texttt{pair} \ : \ A \to B \to A \times B$$

That is, given an element $a : A$ as an argument, and *then* given an element $b : B$ as an argument, `pair` produces an element of $A \times B$, which can be thought of as an ordered pair (a, b). If we continue to think of types as leading a "double life" as both collections and propositions, if A and B are interpreted as propositions, then $A \times B$ can be interpreted as the assertion "A and B", for to construct an element of $A \times B$, both an element of A and an element of B must be supplied as inputs. The product type $A \times B$ also has its own recursor $\texttt{rec}_{A \times B}$, which has the following type signature:

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

If you want to create a function from the product type $A \times B$ into some other type C, then you must first supply the type $C : \mathcal{U}$ as an argument to the recursor, and then supply a *function of two arguments* $f : A \to B \to C$, which tells us how to "transform" an element $a : A$ and an element $b : B$ into an element $c : C$. The recursor $\texttt{rec}_{A \times B}$ transforms this two-argument function into a function $g : A \times B \to C$ which acts on *pairs* (a, b) as opposed to a function that takes arguments one at a time. We can also think of $\texttt{rec}_{A \times B}(C)$ as a function which converts **uncurried** functions $A \to B \to C$ into **curried** functions $A \times B \to C$.

Let's look at a few functions defined using this recursor. For one, we can define two **projection functions** with the type signatures

$$\texttt{pr1} \ : \ A \times B \to A$$

$$\texttt{pr2} \ : \ A \times B \to B$$

which map a pair (a, b) to its first or its second element, respectively. To define these functions, we must first consider how they act on pairs in an *uncurried* way:

$$(a, b) \mapsto a$$

$$(a, b) \mapsto b$$

If these functions were to take the arguments a, b one at a time, rather than "packaged together" in a pair, they would look something like this:

$$a \mapsto (b \mapsto a)$$

$$a \mapsto (b \mapsto b)$$

Or, to describe these actions using lambda functions, we could write

$$(\lambda a.\ \lambda b.\ a)$$

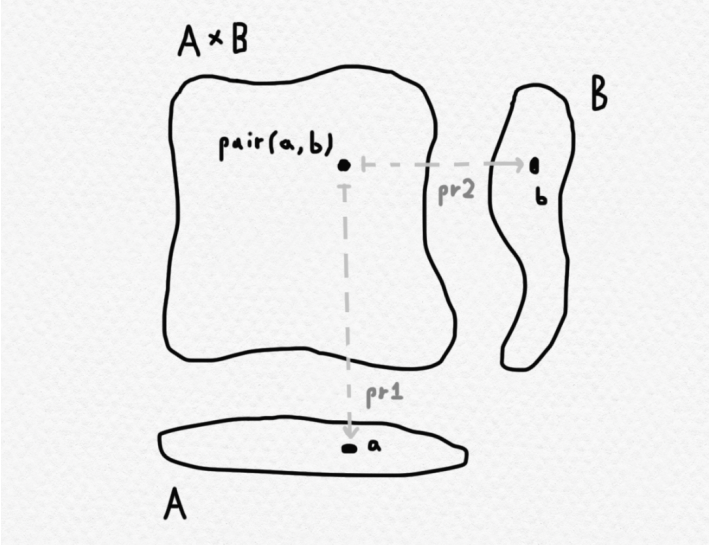
$$(\lambda a.\ \lambda b.\ b)$$

These *curried* versions of the projection functions are functions that we already know how to define using lambda expressions. Hence, to define `pr1` and `pr2`, we can use $\texttt{rec}_{A\times B}$ to "uncurry" these lambda expressions and obtain the desired projectors:

$$\texttt{pr1} = \texttt{rec}_{A\times B}(A, (\lambda a.\ \lambda b.\ a))$$

$$\texttt{pr2} = \texttt{rec}_{A\times B}(B, (\lambda a.\ \lambda b.\ b))$$

As with many other types, there is a kind of geometric/topological way of thinking of the product type $A \times B$ as well: we can think of it as a kind of space whose "axes" are given by A and B, so that every point is determined uniquely by its projection onto A and its projection onto B, like this:



Let's look at another example. Since we can interpret $A \times B$ as a proposition stating "both A and B", we can prove analogues of classical tautologies involving logical conjunction in type theory. For instance, we can prove that "and" is commutative by showing that for any types A, B, we have a function $A \times B \to B \times A$. That is, we can construct an element

$$\texttt{andComm} \ : \ \prod_{A,B:\mathcal{U}} A \times B \to B \times A$$

as a *proof* that logical conjunction is commutative. As a function, `andComm` should transform a pair consisting of an element of A and an element of B into a pair consisting of an element of B and an element of A. The only way of accomplishing this that comes to mind is to map $(a, b) \mapsto (b, a)$. Remember, though, that this is an uncurried function on pairs, and to define this using $\texttt{rec}_{A\times B}$, we must supply a curried version of the same function:

$$a \mapsto (b \mapsto \texttt{pair}(b, a))$$

or

$$\lambda a.\ \lambda b.\ \texttt{pair}(b, a)$$

so that we may write our proof as follows:

$$\texttt{andComm} = \texttt{rec}_{A\times B}(B \times A, (\lambda a.\ \lambda b.\ \texttt{pair}(b, a)))$$

Alternatively, we could define `andComm` by using the preexisting projection functions `pr1` and `pr2` to extract the first and second elements of a pair, rather than using the recursor. We can do this as follows:

$$\texttt{andComm} = \lambda p.\ \texttt{pair}(\texttt{pr2}(p), \texttt{pr1}(p))$$

We can also use product types to define recursive sequences on $\mathbb{N}$ that aren't so straightforward to define using $\texttt{rec}_{\mathbb{N}}$ because each term is defined a function of more than just its immediate predecessor. An example that comes to mind is the *Fibonacci sequence*, defined recursively by the initial values $F_0 = 0, F_1 = 1$ and the recurrence

$$F_{n+1} = F_n + F_{n-1}$$

Because $\texttt{rec}_{\mathbb{N}}$ tells us how to define recursive functions in terms of the index and the previous term, so that $f(n+1) = g(n, f(n))$ for some function g. It is not, however, apparent how to express F_{n+1} as a (simple) function of n and F_n. But if, instead of considering individual Fibonacci numbers, we consider *pairs* of consecutive Fibonacci numbers, we may use the fact that

$$(F_{n+1}, F_n) = (F_n + F_{n-1}, F_n)$$

to express each *pair* as a function of the previous pair:

$$(F_{n+1}, F_n) = g(n, (F_n, F_{n-1}))$$

where

$$g((a, b)) = (a + b, a)$$

Or, defined as a lambda expression,

$$g = \lambda p.\ \texttt{pair}(\texttt{pr1}(p) + \texttt{pr2}(p), \texttt{pr1}(p))$$

Thus, we can define a function $\texttt{fiboPair} : \mathbb{N} \to \mathbb{N} \times N$ that calculates *pairs* of Fibonacci numbers using $\texttt{rec}_{\mathbb{N}}$ as follows:

$$\texttt{fiboPair} = \texttt{rec}_{\mathbb{N}}\Big(\mathbb{N} \times \mathbb{N},\ \texttt{pair}(1, 0),\ \big(\lambda p.\ \texttt{pair}(\texttt{pr1}(p) + \texttt{pr2}(p), \texttt{pr1}(p))\big)\Big)$$

Hence, to extract the nth Fibonacci number from this, we may simply take the second projection of $\texttt{fiboPair}(n)$. Thus, we may define $\texttt{fibo} : \mathbb{N} \to \mathbb{N}$ as follows:

$$\texttt{fibo} = \texttt{pr2} \circ \texttt{fiboPair}$$

2 Coproduct and dependent sum type

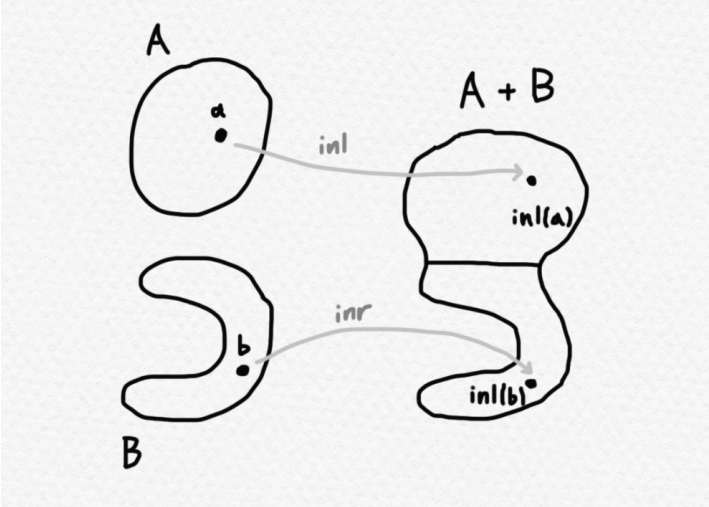
We have seen what the type-theoretic analogues of the logical operators "implies", "not", and "and" are, with the most significant omission so far being "or". Given two types A and B their **coproduct type** or **sum type** is denoted $A + B$, and is also defined as an inductive type. The type $A + B$ has two constructors:

$$\texttt{inl} \ :\ A \to A + B$$
$$\texttt{inr} \ :\ B \to A + B$$

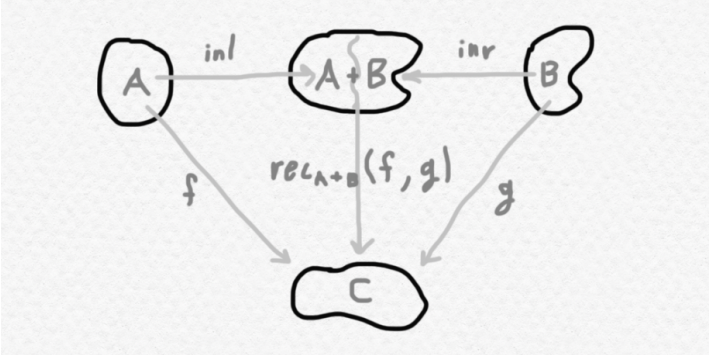
In other words, given an element $a : A$, we can construct an element $\texttt{inl}(a) : A + B$; alternatively, given an element $b : B$, we can construct an element $\texttt{inr}(b) : A + B$. If we have a proof of A, or if we have a proof of B, then we can construct a proof of $A + B$ by applying `inl` or `inr` respectively, hence the logical interpretation of $A + B$ as "A or B". Of course, $A + B$ has a recursor as well, denoted $\texttt{rec}_{A+B}$, which has the following type signature:

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

Given a function $f : A \to C$ and a function $g : B \to C$, the recursor produces a function from $A + B \to C$ which is defined "piecewise", so that it sends $\texttt{inl}(a) \mapsto f(a)$ and $\texttt{inr}(b) \mapsto g(b)$ for all $a : A$ and $b : B$. Pictorially, we can think of $A + B$ as a type consisting of a copy of A and a copy of B, "smushed together":



In fact, if you expand this diagram to include a third type C, and imagine how two functions $f : A \to C$ and $g : B \to C$ are combined using the recursor to create a function $A + B \to C$, and if you've been exposed to a little category theory, you might recognize this as *coproduct diagram*:



Categorically speaking, the recursor $\texttt{rec}_{A+B}$ is defined in such a way that $\texttt{rec}_{A+B}(f, g)$ makes this diagram *commute* for any $f : A \to C$ and $g : B \to C$. (Actually, *coproducts* have a bit of a stronger definition in category theory, and they are a special case of a more general construction called a *colimit*.)

Once again, we can use the inductive definition and constructors of $A + B$ to prove analogues of tautologies from classical logic, such as "A and B implies A or B":

$$A \times B \to A + B$$

A function $\texttt{andImpliesOr} : A \times B \to A + B$, or a proof of this tautology, would take as input a pair (a, b) and return either an element of A included in $A + B$ via `inl`, or an element of B included in $A + B$ via `inr`. Of course, if we are given (a, b), we can use $\texttt{inl}(a) : A + B$ as the desired element. Thus, it suffices to define

$$\texttt{andImpliesOr} = \texttt{inl} \circ \texttt{pr1}$$

Alternatively, given the pair (a, b), we might choose to use $\texttt{inr}(b) : A + B$ rather than $\texttt{inl}(a) : A + B$, and decide to define

$$\texttt{andImpliesOr}' = \texttt{inr} \circ \texttt{pr2}$$

Notice that although these two functions are proofs of the same propositions, they are *not the same function*. They have very different behavior: the former function uses only the first element of any pair given as input, and the latter only uses the second element of any input pair. This is an example of **proof relevance**, i.e. the idea that two proofs of the same proposition can have different "behavior" - in contrast to the concept of **proof irrelevance**, where any two proofs of the same proposition are more or less interchangeable, as is the case in classical first-order logic. This is one of the big

philosophical differences between first-order logic and type-theoretic logic.

There is also a generalization of the sum/coproduct type which allows us to "squish together" an arbitrary family of types, rather than just two. This construction is called the **dependent pair type**. To be specific, given a type $A : \mathcal{U}$ and a type family over A, i.e. a function $B : A \to \mathcal{U}$, the dependent pair type is denoted

$$\sum_{a:A} B(a)$$

and can also be defined inductively with a single constructor called `pair`, which has the following type signature:

$$\texttt{pair} : \prod_{a:A} \left(B(a) \to \sum_{a:A} B(a) \right)$$

That is, given an element $a : A$, and an element of the type $B(a)$ corresponding to this element from the type family B, we can construct an element of the dependent sum type $\Sigma_{a:A}\ B(a)$. Notice that, to construct an element of the dependent sum type, we only need *at least one* of the types $B(a)$ in the type family $B : A \to \mathcal{U}$ to be nonempty. Recall that in order to define an element of the *dependent function type* $\Pi_{a:A} B(a)$, *each* type in the type family must be nonempty, because each element $a : A$ must be mapped into an element of its corresponding type in the type family $B(a)$. Hence, whereas the dependent function type former Π was analogous to the universal quantifier $\forall$ from first-order logic, the dependent pair type former Σ is analogous to the existential quantifier $\exists$ from first-order logic. If we interpret the type family $B : A \to \mathcal{U}$ as a predicate, assigning to each $a : A$ a *proposition* $B(a)$ which can either be true (nonempty) or false (empty), an element of $\Pi_{a:A} B(a)$ states "$B(a)$ is true *for all* $a : A$", but $\Sigma_{a:A} B(a)$ states "$B(a)$ is true *for some* $a : A$", or "there exists $a : A$ such that $B(a)$".

As usual, we also have a recursor for the dependent sum type $\Sigma_{a:A} B(a)$, which we can denote $\texttt{rec}_{\Sigma(a:A)\ B(a)}$. The recursor has the following type signature:

$$\texttt{rec}_{\Sigma(a:A)B(a)} \ : \ \prod_{C:\mathcal{U}} \left(\left(\prod_{a:A} B(a) \to C \right) \to \left(\sum_{a:A} B(a) \right) \to C \right)$$

That is, if we provide a family of functions $f_a : B(a) \to C$ from $B(a)$ to C for each $a : A$, we can assemble them together into a single function from the dependent pair type $\Sigma_{a:A} B(a)$ into C.

Once again, if we interpret the dependent pair type as an analogue of the existential quantifier $\exists$ "there exists", then we can prove analogues of true sentences in first-order logic such as "if A is nonempty and $B(a)$ is true for all a in A, then there exists a in A such that $B(a)$". In first-order logic, this would look like

$$\big((\exists x\ x \in A) \wedge (\forall x\ x \in A \to B(x)) \big) \to (\exists x\ x \in A \wedge B(x))$$

whereas in type-theoretic logic, this looks like

$$A \to \left(\prod_{a:A} B(a) \right) \to \left(\sum_{a:A} B(a) \right)$$

Say we want to write a proof of this proposition, or a function with the above type signature, which we'll call `univToExist`. This function, when given an element $a : A$ and a function $f : \Pi_{a:A} B(a)$ as input, would have to supply an element of $\Sigma_{a:A} B(a)$ as output. To do this, we can just pair up the element a with the element $f(a) : B(a)$ obtained by evaluating f at a, so we may define our function using a lambda expression as follows:

$$\texttt{univToExist} = (\lambda a.\ \lambda f.\ \texttt{pair}(a, f(a)))$$

See the exercises for a couple more examples to try on your own.

You might have noticed that the dependent pair type generalizes the binary sum construction - that is, $A + B$ can be defined as a special case of a dependent pair type. Recall that 2 is the **binary**

type, or the inductive type with two constructors $\texttt{true} : 2$ and $\texttt{false} : 2$ (sometimes they are given other names). If we have two types A, B, we could collect them together into a type family over 2 by defining a type family $F : 2 \to \mathcal{U}$ which maps $\texttt{true} \mapsto A$ and $\texttt{false} \mapsto B$. Then we could define the coproduct type $A + B$ to be the dependent sum $\Sigma_{x:2} F(x)$. To be more specific, 2 has a recursor $\texttt{rec}_2$ with type signature

$$\texttt{rec}_2 \ : \ \prod_{C:\mathcal{U}} C \to C \to (2 \to C)$$

which, given two values $c_0, c_1 : C$, returns a function $2 \to C$ sending $\texttt{true} \mapsto c_0$ and $\texttt{false} \mapsto c_1$. Hence, if we let

$$F = \texttt{rec}_2(A, B)$$

then we could alternatively define $A + B$ as the dependent sum type $\sum_{x:2} F(x)$ so that the constructor $\texttt{inl}$ can be replaced by the function $(\lambda a.\ \texttt{pair}(\texttt{true}, a))$ and the constructor $\texttt{inr}$ can be replaced by the function $(\lambda b.\ \texttt{pair}(\texttt{false}, b))$.

You might also have wondered why the constructor for the dependent sum type is called $\texttt{pair}$, when we already have another constructor for a different inductive type called $\texttt{pair}$, namely the constructor for the product type $A \times B$. This is because the *product type is also a special case* of the dependent pair type: in particular, $A \times B$ can be thought of as a dependent sum of a *constant type family* over A, i.e. the type family sending $a \mapsto B$ for each $a : A$.

3 Teaser on equality types

Any theory of mathematics worth its salt should clearly have a way of expressing equations, or expressing that two expressions have the same value. In first-order logic, this is accomplished by the equality sign $=$, which is defined in terms of certain properties such as reflexivity, symmetricity, and transitivity. In type-theoretic logic as we have developed it so far, types and propositions have been *one and the same*. To continue this pattern, if we want to make assertions involving equalities, these should *also* be types. For instance, the assertion $(\texttt{zero} + \texttt{one} = \texttt{one})$ should itself be a type, and it should therefore have elements, be capable of acting as the domain and codomain of functions, and so on. But what on earth should an element of such a type look like? As a bit of a spoiler, the rich theory of equality types in type theory is where the name *Homotopy* Type Theory comes from. We shall see later that topological analogies are very helpful in understanding how equality types behave.

Given a type A, we have a parametrized family of types $A \to A \to \mathcal{U}$ defining the *equality types* of A. Given $x, y : A$, we write $(x =_A y)$, or sometimes just $(x = y)$ (when A is clear from context) to denote the **equality type** of x and y. If x and y are indeed the same element of A, this type will be nonempty, representing a true proposition. If x and y are distinct, the equality type $(x =_A y)$ will be an empty type. Actually, the family of equality types are inductively defined, with a single constructor called $\texttt{refl}$. This constructor has the following type signature:

$$\texttt{refl} : \prod_{a:A} (a =_A a)$$

Given an element of A, $\texttt{refl}(a)$ asserts that a is equal to a, hence the name $\texttt{refl}$, short for "reflexivity". There is also what is called an *induction principle* for equality types, a generalization of recursion principles that allows us to define dependently types functions out of equality types. We haven't yet discussed induction principles (actually, all of the inductive types have one, we just haven't talked about them yet), but we'll talk about them first thing in the next session. Notice that we haven't *assumed* the symmetric and transitive properties of equality as part of the definition - these are actually *theorems* that can be proven from the induction principle. More on this later!

4 Exercises and teasers

Now we can finish filling in the table of analogous concepts from first-order logic and type-theoretic logic that was left incomplete in the last write-up. However, I've added a few more rows at the bottom

with a few concepts that we haven't found analogues for yet. Can you think of any ways to complete the table? What would the HoTT analogues of these concepts look like?

First-order logic	Type-theoretic logic
Proposition	Type
Proof of a proposition	Element/inhabitant of a type
True proposition	Inhabited type (e.g. $\mathbb{1}$)
False proposition	Empty type (e.g. $\mathbb{0}$)
Implication $A \to B$	Function type $A \to B$
Modus ponens	Function evaluation
Negation $\neg A$ or "A implies false"	Function type $A \to \mathbb{0}$
Conjunction, A and B	Product type, $A \times B$
Disjunction, A or B	Coproduct type, $A + B$
Predicate P on set A	Type family $P : A \to \mathcal{U}$
Universal quantifier $\forall$	Product type former Π
For all $x \in A$, $P(x)$	$\prod_{x:A} P(x)$
Existential quantifier $\exists$	Dependent pair type former Σ
There exists $x \in A$ such that $P(x)$	$\Sigma_{a:A} P(a)$
Equality $x = y$ or "x equals y"	Equality type $x =_A y$?
Reflexive property of $=$	Constructor `refl`
Symmetric property of $=$	???
Transitive property of $=$	???
Axiom of choice	???
Induction on $\mathbb{N}$	???

Here are some exercises to think about if you want to play with type theory before the next seminar. Some of them are taken from the HoTT book, which you can find a full PDF of online.

1. Prove that if there exists $a : A$ such that $B(a)$, then it is not true that $\neg B(a)$ for all $a : A$. In other words, construct an element (proof) with type signature

$$\Big(\sum_{a:A} B(a)\Big) \to \neg\Big(\prod_{a:A} \neg B(a)\Big)$$

   for a given type A and type family $B : A \to \mathcal{U}$.

2. The **Law of Excluded Middle** is the following type:

$$\texttt{LEM} = \prod_{A:\mathcal{U}} (A + \neg A)$$

   which states, in plain english, "for every proposition A, either A is true or A is false" or perhaps "for every type A, either A is empty or A is nonempty". It turns out that it is impossible to construct an element of `LEM`, nor is it possible to construct an element of $\neg$`LEM`. (Take my word for it, or try to construct an element yourself until you give up.) However, the following modification of `LEM` is provable:

$$\prod_{A:\mathcal{U}} \neg\neg(A + \neg A)$$

Construct an element of this type, i.e. a proof that for any proposition A, it is *not false* that either A is true or A is false.

3. Prove the tautology "if not (A or B), then (not A) and (not B)". That is, supply an element of the type
$$\neg(A + B) \to (\neg A \times \neg B)$$
for any given types A, B.

4. Given functions $\texttt{add} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$ and $\texttt{mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, define a function $\texttt{isPrime} : \mathbb{N} \to \mathcal{U}$ which maps each natural number $n : \mathbb{N}$ to either the empty type $\mathbb{0}$ if n is not prime, or the unit type $\mathbb{1}$ if n is prime. You can define a few intermediate functions if necessary.