Homotopy Type Theory Seminar Notes (Session 1)

Franklin

July 13, 2022

1 Propositions as types

In type theory, we have objects called **types** which behave, intuitively, somewhat like sets in set theory. Like sets, types can have **elements** or **inhabitants**, and to express that an object *a is (an element) of type A*, we write $a : A$. In type theory, however, a type leads a "double life" not only as a formalization of the notion of a *collection*, but also as a *proposition.* In set theory, for instance, if we want to make a formal statement about sets, we must phrase the statement in the language of first-order logic, which is separate from the set-theoretic universe. In type theory, on the other hand, the way we express propositions is *as types themselves.* For example, if in set theory we wanted to assert the existence of a set A such that any two elements of A are equal, we would write the following first-order sentence:

$$\exists A\ \forall x\ \forall y\ (x \in A)(y \in A) \to (x = y)$$

In type theory, on the other hand, if we wanted to assert the existence of a type A such that any two inhabitants of A are equal, this claim itself would be a type:

$$\sum_{A:\mathcal{U}} \prod_{x:A} \prod_{y:A} (x =_A y)$$

A proof of this proposition would not only have to tell us that the proposition is true, but it would in fact supply such a type A, as well as a dependent function taking two arguments $x, y : A$ and returning a proof of $x = y$. In general, proofs in type theory tend to be more useful by virtue of being **constructive**, so that when they assert the existence of some object with such and such property, they also tell us exactly how to construct that object, using, for example, a dependently typed function that allows us to produce objects with the claimed property on demand. The above type signature uses machinery such as the dependent sum type Σ and the equality type $=_A$ that we didn't get to in this session, but hopefully we will cover both of these concepts in the next session.

In our correspondence between types and propositions, **inhabited types**, i.e. types containing some inhabitant, represent *true propositions*, and **empty/uninhabited types** (such as **the empty type** denoted $\mathbb{0}$) represent false propositions. In this way, an element $a : A$ of a type A serves as "evidence" or "proof" for the proposition that A represents, since knowing that A has an inhabitant tells us that it is inhabited, and therefore true as a proposition. In order for our type-theoretic logic to have all the expressive power of first-order logic, we would like to be able to express more complex propositions like $A \to B$ "A implies B", $\neg A$ "not A", $A \wedge B$ "A and B", and so on in terms of simpler propositions A, B. These concepts, of course, all have type-theoretic analogues, but in this session we only discussed implication $\to$ and negation $\neg$.

For any two types A and B, we may construct a **function type** denoted $A \to B$, which is the type whose elements are the functions mapping each element of A to an element of B. One way of actually constructing functions of type $A \to B$ is using **lambda expressions**, which directly describe how a function acts on its inputs. For instance, for any type A, we may define the **identity function** $\mathrm{id}_A : A \to A$ using the following lambda expression:

$$\mathrm{id}_A = (\lambda x.\ x)$$

which simply asserts that id_A takes a single input x, and returns the value of that input. We can also define two functions of the function type $A \to (A \to A)$ (which means the same thing as $A \to A \to A$) as follows:

$$\text{first}_A = (\lambda x.\ \lambda y.\ x)$$

$$\text{second}_A = (\lambda x.\ \lambda y.\ y)$$

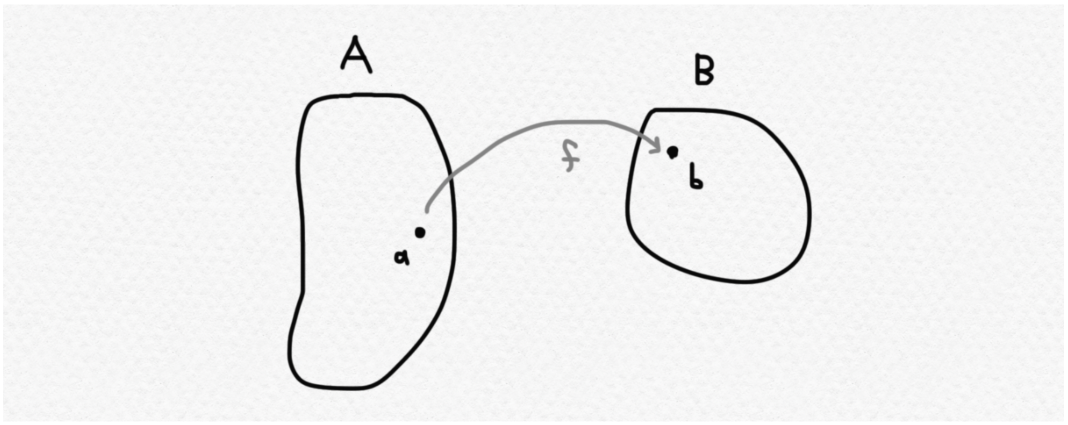
The function first_A takes an argument $x : A$, then takes an argument $y : A$, then returns x - in other words, it takes two arguments and returns its first argument. On the other hand, second_A takes an argument $x : A$, then takes an argument $y : A$, then returns its second argument y. Notice that you can think of a function $f : A \to A \to A$ in one of two ways:

1. You could think of f as a function with two arguments of type A, and an output of type A. For instance, first_A takes two arguments x, y and returns x, and second_A takes two arguments x, y and returns y.

2. You could also think of f as a function with one argument of type A, and an output of type $A \to A$ - in other words, a function that takes an element of A as input, and returns a function from A to A as output. Under this interpretation, first_A takes one argument x and returns the constant function $A \to A$ which always returns x, and second_A takes one argument x and returns the identity function id_A. In fact, second_A could be equivalently defined as

$$\text{second}_A = (\lambda x.\ \text{id}_A)$$

So if all types also stand for propositions, and A, B are two types representing some propositions, what does $A \to B$ signify propositionally? Well, recall that A is "true" if and only if it has an inhabitant $a : A$. If $A \to B$ is also true, then there must be a function $f : A \to B$, and if we have both a function $f : A \to B$ and an element $a : A$, then we can evaluate the function f at the argument a to obtain an element $f(a) : B$. Therefore, if *if* A is true *and* $A \to B$ is true, *then* B is true. This means that the function type $A \to B$ behaves just like the implication $A \to B$ "A implies B" from propositional logic! Additionally, **modus ponens**, the elimination rule in propositional logic that allows us to infer B from A and $A \to B$, is analogous to evaluating a function from $A \to B$ at an argument from A to obtain an element of type B. We can also think of a function $f : A \to B$ as a way of "turning proofs of A into proofs of B".



Now that we have found a type-theoretic way of expressing implication, let's use it to come up with an analogue of negation. Saying "not A" is essentially the same as saying that A is impossible, or to say the same thing more obliquely, "A implies false". But we've already found an analogue for implication, as well as an analogue for falsity, namely an empty type like $\mathbb{0}$. Thus, the proposition "not A" can be expressed as *the function type from A into the empty type* $A \to \mathbb{0}$. Intuitively speaking, this makes sense because if there exists a function $f : A \to \mathbb{0}$, then it cannot be the case that A is inhabited, for if $a : A$ were an element of type A, then $f(a)$ would be an element of the empty type $\mathbb{0}$.

Using this correspondence, we can state tautologies from classical logic in type-theoretic logic, and prove them by defining functions using lambda expressions. For instance, to prove the tautology

$A \to A$, we may use the identity function $\mathrm{id}_A : A \to A$. For a more interesting example, we can try proving the following tautology:

$$A \to ((A \to \mathbb{0}) \to \mathbb{0})$$

or, using the definition of negation,

$$A \to \neg\neg A$$

which, in plain English, says that "A implies not not A". To prove this tautology, we can define the following lambda function:

$$\mathrm{dneg}_A = (\lambda x.\ \lambda f.\ f(x)) \ :\ A \to \neg\neg A$$

Given an element $x : A$ and a function $f : A \to \mathbb{0}$, we could evaluate f at x to obtain an element $f(x) : \mathbb{0}$. Notice that the function dneg_A can never be fully evaluated, for if it were, it would produce an element of $\mathbb{0}$, which is impossible - but when *partially evaluated*, it produces an element of $\neg\neg A$, which may very well be inhabited. As another example, we can prove the tautology

$$(A \to B \to C) \to (B \to A \to C)$$

for arbitrary types A, B, C using the following lambda expression:

$$\mathrm{swap} = (\lambda f.\ \lambda b.\ \lambda a.\ f(a, b))$$

Notice that we can think of swap as proof of the aforementioned tautology, or we can think of it as a function which accepts a function $f : A \to B \to C$ as an input, and returns another function $g : B \to A \to C$ which is the same as f except that it accepts its inputs in reverse order.

2 Type families and dependent function types

In classical first-order logic, not only do we have propositions that can be either true or false, but we also have **predicates** which are like propositions that contain variables, whose truth value can very depending on the value of those variables. For instance, an example of a predicate P in the theory of Peano Arithmetic would be the following:

$$P(x) = \big(\exists y\ (1+1)\cdot y = x\big)$$

which can be interpreted as stating that x is an even number, i.e. there exists y such that $2y = x$. P does not have a single truth value, but rather it can take a different truth value depending on the value of x - for instance $P(1)$ is false, $P(2)$ is true, $P(3)$ is false again, and so on.

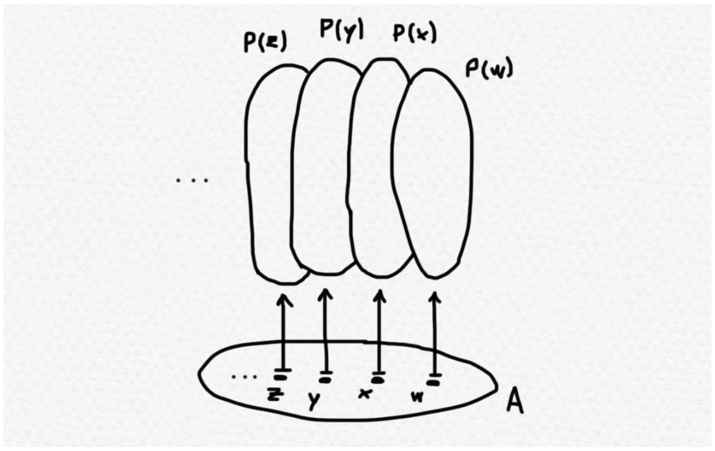
Predicates also have an analogue in type theory, called **type families**. We can think of a predicate P as a function from an *indexing set* A (in the former example, $\mathbb{N}$) into a set of propositions, each of which can be true or false. Since types and propositions are identified in type theory, a type family P should assign each element $a : A$ of the *indexing type* A to *another type* denoted $P(a)$, and each of the types $P(a)$ can be either empty or nonempty depending on whether the proposition represented by $P(a)$ is true or false. Thus, a type family, if it is to be analogous to a predicate, should map each element of A to another type - hence, it should be a function from A into the type of all types.

In case the phrase "type of all types" makes your alarm bells go off (as it should!), here's a better description of what this actually means. In type theory, there is a hierarchy of type-theoretic **universes**

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \mathcal{U}_3 : \cdots$$

each of which is an inhabitant of the subsequent universe. We would like to have a "domain of discourse" in which to do all of our type-theoretic constructions, but if we attempted to define a "type of all types", we would quickly arrive at a contradiction similar to Russel's Paradox. Thus, we will do most of our constructions within one of these universes $\mathcal{U}_i$, but if we wanted to do some reasoning about the universe $\mathcal{U}_i$ itself we would have to "move up one level" into the higher-level universe $\mathcal{U}_{i+1}$. This "fix" for Russel's Paradox is similar to the use of proper classes (i.e. collections that are "too big" to be sets) in NBG set theory, as a way of manipulating large collections of sets without running into paradoxes.

Therefore, if we're working inside some universe $\mathcal{U}$, a type family P over A (or *indexed by* A) should assign each element of A to a type in the universe, so that it should be a function $P : A \to \mathcal{U}$. This is often represented pictorially as follows:



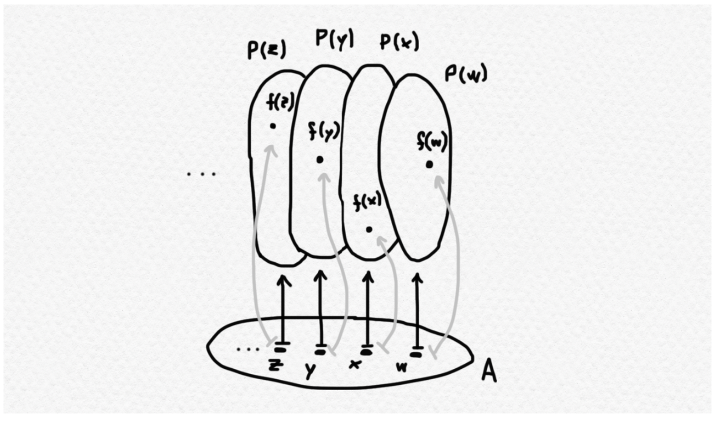
(You may find similar pictures in category-theoretic descriptions of *bundles*, for instance in Goldblatt's *Topoi*). Now that we know what a type family is, we are ready to discuss **dependently typed functions**. Given two types A, B, the function type $A \to B$ discussed earlier consists of functions that send all elements of A to elements *of the same type* B. A dependently typed function, however, may send two different inputs $x, y : A$ to outputs of different types. In particular, given a type family $P : A \to \mathcal{U}$, we may construct the **dependent product type**

$$\prod_{x:A} P(x)$$

which is the type whose inhabitants are dependently typed functions f such that for all $x : A$, the evaluation $f(x)$ has type $P(x)$. We can augment our earlier picture to show how a dependently typed function

$$f : \prod_{(x:A)} P(x)$$

acts on the elements of A:



Notice that if we are able to actually define a function $f : \prod_{(x:A)} P(x)$, then we know that $P(x)$ is any inhabited type for all $x : A$ - namely, it has $f(x)$ as one of its inhabitants. Therefore, interpreted as propositions, we have that $P(x)$ is *true* for all $x : A$. Thus, the dependent product type behaves just like the universal quantifier $\forall$ "for all" in first-order logic: an element f of the dependent product type $\prod_{(x:A)} P(x)$ is proof that $P(x)$ is true for all $x : A$, and if we want to "weaken" this proof to obtain a proof of $P(a)$ for some *specific* $a : A$, all we need to do is evaluate our function f at a to obtain the desired proof $f(a) : P(a)$.

We can use dependent product types to generalize the proofs of propositional tautologies that we wrote earlier. For some *given* type A, the function $\mathrm{dneg}_A : A \to \neg\neg A$ expresses that A implies its double negation, but we can also consider the more powerful claim given by the type

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

Intuitively, $A \to \neg\neg A$ states "A implies not not A" for the specific type A, but the product type $\prod_{(A:\mathcal{U})}(A \to \neg\neg A)$ states "*for all propositions (types)* A in the universe, A implies not not A". To construct an element of this type, we can also use a lambda function:

$$\mathrm{dneg} = \big(\lambda A.\ \lambda x.\ \lambda f.\ f(x)\big) : \prod_{(A:\mathcal{U})} (A \to \neg\neg A)$$

so that the partial application $\mathrm{dneg}(A)$ is the same as the dneg_A that we defined earlier.

We may also define a useful function of the following type:

$$\mathrm{compose} : \prod_{A,B,C:\mathcal{U}} \big((A \to B) \to (B \to C) \to (A \to C)\big)$$

using the following lambda expression:

$$\mathrm{compose} = (\lambda A.\ \lambda B.\ \lambda C.\ \lambda g.\ \lambda f.\ \lambda x.\ f(g(x)))$$

so that for functions $g : A \to B$ and $f : B \to C$, we have that

$$\mathrm{compose}(A, B, C, g, f) \ : \ A \to B$$

is the **composite function** $f \circ g$. We will often use this well-known functional composition notation rather than writing $\mathrm{compose}(A, B, C, g, f)$ for the sake of conciseness.

3 Inductive types and recursors

Much of the interesting mathematics done in type theory is implemented using *inductive types*. An **inductive type** I is a special kind of type that is described in terms of all the possible ways of constructing elements of that type, called its **constructors**, as well as a way of constructing functions of the form $I \to A$, called the **recursion principle** for that type (there is also a way of constructing *dependent functions* out each inductive type, but we will discuss this later). One of the simplest examples of an inductive type is the **unit type** denoted $\mathbb{1}$, which is defined by the following constructor:

$$* \ : \ \mathbb{1}$$

We can think of $\mathbb{1}$ as a type containing only one element, because there is only one way of obtaining an element of $\mathbb{1}$, namely from its unique constructor $*$. As a warning, however, keep in mind that the fact that $*$ is the unique inhabitant of $\mathbb{1}$ (or, equivalently, that all elements of $\mathbb{1}$ are equal to $*$) is *not* part of the definition of $\mathbb{1}$ - it is a *theorem* that we will be able to prove later after learning about equality types. The inductive type $\mathbb{1}$ also comes equipped with a recursion principle $\mathbb{1}$ of the following type signature:

$$\mathrm{rec}_{\mathbb{1}} \ : \ \prod_{(A:\mathcal{U})} \big(A \to (\mathbb{1} \to A)\big)$$

Given a type $A : \mathcal{U}$ and an element $a : A$ as inputs, the recursor $\mathrm{rec}_{\mathbb{1}}$ produces an element of the type $\mathbb{1} \to A$, specifically a function $f : \mathbb{1} \to A$ such that $f(*) = a$. In other words, to define a function with domain $\mathbb{1}$ using the recursor, we only need to specify the value of $f(*)$ - so rather than explicitly assuming that $*$ is the only element of $\mathbb{1}$, we kind of *implicitly* assume it by making it possible to define a function $\mathbb{1} \to A$ by only specifying its value at $*$. Keep this idea in mind - it will be conceptually useful when we actually prove later on that $* : \mathbb{1}$ is the unique element of the unit type.

For a less trivial example of an inductive type, let us consider how the **natural numbers** are defined type-theoretically. The natural numbers are an inductive type with the following two constructors:

$$\text{zero} \ : \ \mathbb{N}$$
$$\text{succ} \ : \ \mathbb{N} \to \mathbb{N}$$

In other words, the only two ways of producing a natural number are to use zero, or to evaluate succ with some other preexisting natural number as its input. Again, we do not *assume* that the only elements of $\mathbb{N}$ are the values

$$\text{zero}, \ \text{succ}(\text{zero}), \ \text{succ}(\text{succ}(\text{zero})), \ \cdots$$

nor are we even assuming that these values are actually distinct. These are all facts that we will be able to prove later on using equality types. Rather, we assume the existence of a recursor $\text{rec}_{\mathbb{N}}$ with the type signature

$$\text{rec}_{\mathbb{N}} : \prod_{(A:\mathcal{U})} \big(\mathbb{N} \to (\mathbb{N} \to \mathbb{A} \to \mathbb{A}) \to (\mathbb{N} \to \mathbb{N})\big)$$

such that $\text{rec}_{\mathbb{N}}(A, x, g) = f : \mathbb{N} \to A$ is a function satisfying the equations

$$f(\text{zero}) = x$$
$$f(\text{succ}(n)) = g(n, f(n))$$

for any $n : \mathbb{N}$. In other words, in addition to the use of lambda functions, we are able to define functions $f : \mathbb{N} \to A$ by specifying an initial value or "base case" $x = f(\text{zero}) : A$, and providing a function $g : \mathbb{N} \to A \to A$ that tells us how to calculate $f(\text{succ}(n))$ in terms of the index n and the previous function value $f(n)$.

To practice using the recursor for $\mathbb{N}$, let's try to define a function double : $\mathbb{N} \to \mathbb{N}$ that doubles natural numbers. Notice that double can be calculated recursively using the fact that the double of a number's successor is the successor of the successor of its double - that is, if we increment the input value once, the output gets incremented twice. In other words,

$$\text{double}(\text{succ}(n)) = \text{succ}(\text{succ}(\text{double(n)}))$$

or we could also write

$$\text{double}(\text{succ}(n)) = g(n, \text{double}(n))$$

where

$$g = (\lambda n.\ \lambda d.\ \text{succ}(\text{succ}(d)))$$

Note that this is not enough information to specify the double function - we must also provide a "base case", or the value of double(zero). We will, of course, want to set double(zero) = zero, but we could very easily use a different value for the base case, such as succ(zero) or 1, in which case we would obtain the function $x \mapsto 2x + 1$ rather than $x \mapsto 2x$. Finally, we are ready to package together a complete definition of double using the recursor:

$$\text{double} = \text{rec}_{\mathbb{N}}\big(\mathbb{N}, \text{zero}, (\lambda n.\ \text{succ} \circ \text{succ})\big)$$

For a trickier example, let's try to define addition on the natural numbers. We want a function with the following type signature:

$$\text{add} \ : \ \mathbb{N} \to \mathbb{N} \to \mathbb{N}$$

We usually think of addition as a function of two arguments that returns a natural number, but in this case it is more useful to this of addition as a function of *one argument* whose output is itself a function

$\mathbb{N} \to \mathbb{N}$. In particular, the partially applied function add(zero) : $\mathbb{N} \to \mathbb{N}$ should be the function that adds zero to its input, and the function add(succ(zero)) : $\mathbb{N} \to \mathbb{N}$ should return its input plus one, and so on. In general, add(n) : $\mathbb{N} \to \mathbb{N}$ should be the function that adds n to its input.

To define add recursively, we need to determine a function g that defines add(succ(n)) in terms of add(n). If we already have the function add(n) at our disposal, how can we obtain the function add(succ(n)) from it? Well, adding $(n+1)$ to a natural number is the same as adding n and then adding 1, or adding n and then taking the successor of the result. Therefore, we might want to use the following identity:

$$\mathrm{add}(\mathrm{succ}(n), m) = \mathrm{succ}(\mathrm{add}(n, m))$$

or equivalently we can use the partially-applied version of this equation:

$$\mathrm{add}(\mathrm{succ}(n)) = \mathrm{succ} \circ \mathrm{add}(n)$$

so that we may write

$$\mathrm{add}(\mathrm{succ}(n)) = g(n, \mathrm{add}(n))$$

where

$$g = (\lambda n.\ \lambda a.\ \mathrm{succ} \circ a)$$

Now we just need to figure out the base case add(zero). But this is easy - zero is defined to be the additive identity, so that adding zero to a natural number leaves the input unchanged. Thus, we may simply let add(zero) be $\mathrm{id}_{\mathbb{N}}$, the identity function on $\mathbb{N}$! Thus, we can compile all of these properties into a recursive definition of addition on the natural numbers:

$$\mathrm{add} = \mathrm{rec}_{\mathbb{N}}\big(\mathbb{N} \to \mathbb{N}, \mathrm{id}_{\mathbb{N}}, (\lambda n.\ \lambda a.\ \mathrm{succ} \circ a)\big)$$

Now let's briefly take a look at some less familiar inductive types. For instance, let's define an inductive type called BTree whose elements can be interpreted as binary trees. The type will be defined using the following two constructors:

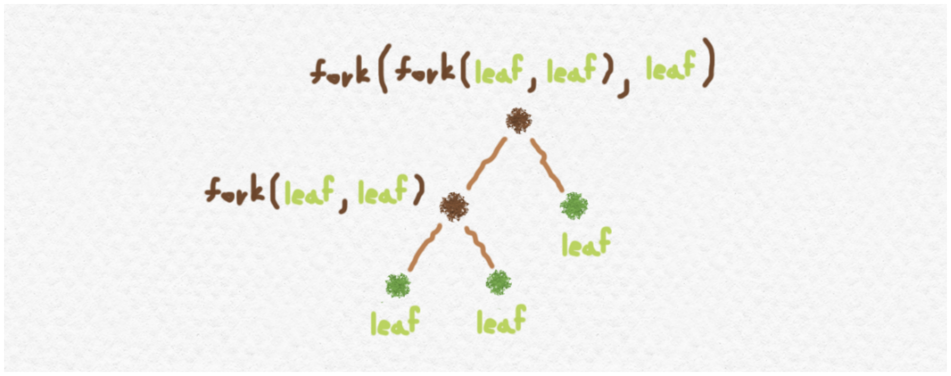
$$\mathrm{leaf}\ :\ \mathrm{BTree}$$

$$\mathrm{fork}\ :\ \mathrm{BTree} \to \mathrm{BTree} \to \mathrm{BTree}$$

The element leaf is meant to be interpreted as a binary tree with only one node (a single leaf), and given two preexisting binary trees T, T', the constructor fork can be used to construct a new binary tree fork(T, T') by "joining together" the trees T, T' at a root node. For instance, the expression

$$\mathrm{fork}(\mathrm{fork}(\mathrm{leaf}, \mathrm{leaf}), \mathrm{leaf})$$

can be visualized as the following binary tree:



Intuitively speaking, we can define a function BTree $\to A$ on a "case-by-case basis" by defining how it acts on inputs taking the form leaf, and how it acts on inputs taking the form fork(T, T'). Formally, however, we can describe how to define functions BTree $\to A$ by describing the recursor

$$\mathrm{rec}_{\mathrm{BTree}} \;:\; \prod_{A:\mathcal{U}} \big(A \to (\mathrm{BTree} \to \mathrm{BTree} \to A \to A \to A) \to (\mathrm{BTree} \to A)\big)$$

which, when supplied with an initial value $x : A$ and a function $g : \mathrm{BTree} \to \mathrm{BTree} \to A \to A \to A$, returns a function $f : \mathrm{BTree} \to A$ such that

$$f(\mathrm{leaf}) = x$$

$$f(\mathrm{fork}(T, T')) = g(T, T', f(T), f(T'))$$

For instance, we could define a function $f : \mathrm{BTree} \to \mathbb{N}$ which takes a binary tree as input, and returns as output a natural number which counts how many leaves the tree has. Let's call this function leafcount : BTree $\to \mathbb{N}$ and try to define it using the recursor. First, we must specify the value that it takes at leaf, which we clearly want to equal 1, or succ(zero), since leaf is a kind of "trivial binary tree" with one leaf. Next, we need to specify the value of leafcount(fork(T, T')) in terms of T, T' and leafcount(T), leafcount(T'). To determine the leaf count of a binary tree with a fork at its root, it suffices to count the number of leaves on the left half, and then count the number of roots on the right half, and then add up the two resulting values. Hence, we want

$$\mathrm{leafcount}(\mathrm{fork}(T, T')) = \mathrm{add}(\mathrm{leafcount}(T), \mathrm{leafcount}(T'))$$

or

$$\mathrm{leafcount}(\mathrm{fork}(T, T')) = g(T, T', \mathrm{leafcount}(T), \mathrm{leafcount}(T'))$$

where g is given by

$$g = (\lambda t.\ \lambda t'.\ \mathrm{add})$$

Therefore, we can define leafcount using the recursor as follows:

$$\mathrm{leafcount} = \mathrm{rec}_{\mathrm{BTree}}\big(\mathbb{N}, \mathrm{succ}(\mathrm{zero}), (\lambda t.\ \lambda t'.\ \mathrm{add})\big)$$

If you want, you can practice using recursors some more in the exercises. However, just one more note before the end of this write-up: the empty type $\mathbb{0}$, which we mentioned earlier, is actually defined as an inductive type *with no constructors*, so that there is *no way* of constructing an element of type $\mathbb{0}$. Accordingly, the recursor $\mathrm{rec}_{\mathbb{0}}$ has the type signature $\prod_{(A:\mathcal{U})}(\mathbb{0} \to A)$, for given any type A, we can always define a function $\mathbb{0} \to A$ without supplying any additional data!

4 Exercises and teasers

Here's a table showing the correspondence between corresponding concepts in first-order logic and type-theoretic logic. A few entries of the table are left blank, because they include concepts that we won't discuss until the next session. However, if you're curious, you can think about what sorts of constructions might be analogous to these logical constructs.

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	???
Disjunction, A or 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 $\exists$	???
There exists $x \in A$ such that $P(x)$	???
Equality $x = y$ or ”x equals y”	???

Now, 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 the law of the contrapositive, i.e. that $(A \to B) \to (\neg B \to \neg A)$ for any types $A, B : \mathcal{U}$.

2. **(HoTT 1.11)** Prove that $\neg\neg\neg A \to \neg A$ for any type $A : \mathcal{A}$. Interpreted as a logical statement, what does this say?

3. Prove that
$$\left(\left(\prod_{A:\mathcal{U}}(\neg\neg A \to A)\right) \to B\right) \to \neg\neg B$$
for any type $B : \mathcal{U}$. Interpreted as a logical statement, what does this say?

4. **(HoTT 1.8)** Define multiplication and exponentiation of natural numbers using $\mathrm{rec}_{\mathbb{N}}$.

5. Let's define an inductive type that models binary trees whose leaves can be one of two colors, either red or blue. Our inductive type, called RedBluBTree, will have the following constructors:
$$\mathrm{redLeaf} : \mathrm{RedBluBTree}$$
$$\mathrm{bluLeaf} : \mathrm{RedBluBTree}$$
$$\mathrm{fork} : \mathrm{RedBluBTree} \to \mathrm{RedBluBTree} \to \mathrm{RedBluBTree}$$

   (a) State the type signature of the recursor $\mathrm{rec}_{\mathrm{RedBluBTree}}$ and the recursive equations satisfied by the functions it returns. You should use this recursor for the subsequent parts of this exercise.

   (b) Define a recursive function $\mathrm{swapColor} : \mathrm{RedBluBTree} \to \mathrm{RedBluBTree}$ that takes a red-blue binary tree as input, and returns a red-blue binary tree with the same structure but opposite colors at each leaf.

   (c) Define a recursive function $\mathrm{redcount} : \mathrm{RedBluBTree} \to \mathbb{N}$ that counts the number of red leaves of a red-blue binary tree.

   (d) Define a recursive function uncolor : RedBluBTree $\to$ BTree that converts a red-blue binary tree into an ordinary binary tree (of the inductive type defined earlier) by "forgetting" the color of each leaf.

6. Can you design your own inductive type that represents a tree in which each non-leaf node can either split into two children *or* three children? (This is called a **2-3 Tree**.) What is the recursor for this type?