Topic 5: Inductive types

May 27, 2014

The notion of *induction* should be familiar from induction proofs over the natural numbers. The basic idea is that in order to prove a statement about all natural numbers, it is sufficient to:

- prove it in the case that the number is zero,
- prove it in the case that the number is the successor of another number for which the statement has already been proven.

Under the propositions-as-types correspondence, this also gives rise to the principle of *recursion*: in order to construct a sequence of elements of a set, it is sufficient to:

- construct an element of the set corresponding to the first (or 'zeroth') element of the sequence,
- construct a function which takes an already constructed element of the sequence as input and outputs the following element of the sequence.

We will see how this induction and recursion correspond to the elimination rule that we will postulate for the type of natural numbers. In fact, we will see that induction and recursion are not at all only a property of the natural numbers; in fact, there are many *inductive types* for which induction proofs and recursive constructions make sense.

In fact, we have already encountered many examples of inductive types: most of the types that we have considered so far can be regarded as inductive types, assuming that one uses a sufficiently general definition of what an inductive type is. This statement explains the notation 'ind$(\ldots)$' for elements obtained via application of an elimination rule. Even universes are a kind of inductive type, when defined in a slightly different way (Tarski-style). There are so many schemes which formalize the idea of an inductive type at a very high level of generality, like inductive-recursive and inductive-inductive types, that we can impossibly discuss all of these. Hence we limit ourselves to giving an informal idea of what inductive types and their various generalizations are about and refer to the research literature for more details. We will meet one generalization of inductive types, namely *higher inductive types*, later on as a new kind of construction specific to *homotopy* type theory.

The natural numbers as an inductive type. An inductive type is specified by a collection of *constructors* which tell us how to obtain elements of the type under consideration. As a running example, we consider the type $\mathbb{N}$ of natural numbers, for which there are two constructors:

- $0 : \mathbb{N}$.

- $\mathtt{succ} : \mathbb{N} \to \mathbb{N}$.

In words: 0 is a natural number, and whenever x is a natural number, then so is $\mathtt{succ}(x)$, which stands for the successor of x. The important point here is that both constructors return a natural number. Intuitively, the inductive type $\mathbb{N}$ is then specified as 'freely generated' by these two constructors in the sense that the elements of $\mathbb{N}$ correspond precisely to all those expressions which can be formed by repeatedly applying these constructors:

$$0, \quad \mathtt{succ}(0), \quad \mathtt{succ}(\mathtt{succ}(0)), \quad \mathtt{succ}(\mathtt{succ}(\mathtt{succ}(0))), \quad \ldots$$

This intuition applies to any inductive type: roughly speaking, its elements are the expressions formed by repeatedly applying the constructors, where a constructor may take any number of arguments from the inductive type itself.

As is always in type theory, this intuitive explanation of what the elements of the type $\mathbb{N}$ are does not constitute a definition of $\mathbb{N}$ in type theory, since types cannot be defined via their elements. The definition of $\mathbb{N}$ is rather given by specifying a set of new inference rules which refer to $\mathbb{N}$. In line with the idea of making our exposition more and more informal, while the reader should retain the ability to fill in all formal details if necessary, we state these rules now in a semi-informal manner, which has the advantage of being more comprehensible to a human than the inference rules written out in all detail.

1. Formation rule: $\mathbb{N} : \mathcal{U}$ in any context Γ.
2. Introduction rules:
   - $0 : \mathbb{N}$ in any context.
   - In a context in which $x : \mathbb{N}$, also $\mathtt{succ}(x) : \mathbb{N}$.
3. Elimination rule: In order to construct an element of $C(x)$ for a dependent type $C : \mathbb{N} \to \mathcal{U}$ and a given $x : \mathbb{N}$, it is enough to construct an element of $z : C(0)$ and for every $n : \mathbb{N}$, a function $f(n) : C(n) \to C(\mathtt{succ}(n))$. This data gives rise to $\mathrm{ind}(C, z, f, x) : C(x)$.
4. Computation rule: In terms of the data used in the elimination rule, we have

$$\mathrm{ind}(C, z, f, 0) \equiv z : C(0), \qquad \mathrm{ind}(C, z, f, \mathtt{succ}(x)) \equiv f(x, \mathrm{ind}(C, z, f, x)) : C(\mathtt{succ}(x)).$$

As for all inductive types, the basic idea behind the elimination and computation rules is this: in order to prove a statement about a particular natural number x, it is enough to prove it for the 'base cases', i.e. for those elements of the type which are obtained by applying the introduction rule; since x is arbitrary, this can also be interpreted as proving a statement about *all* natural numbers. In case that this introduction rule takes one or more arguments from the type itself, then corresponding induction hypotheses may be assumed. If one plugs in one of the base cases into the elimination rule, then the resulting element is judgementally equal to the given data for that base case.

Again, one can construct a universal instance of the elimination rule, which looks like this:

$$\mathrm{ind}'_{\mathbb{N}} : \prod_{C : \mathbb{N} \to \mathcal{U}} C(0) \to \left(\prod_{n : \mathbb{N}} C(n) \to C(\mathtt{succ}(n)) \right) \to \prod_{x : \mathbb{N}} C(x).$$

As a final remark, let it be noted that the introduction of $\mathbb{N}$ has modified the type theory, since we have introduced a new set of inference rules. Generally, introducing a new inductive type always modifies the theory due to the introduction of new inference rules. From a different point of view, which is the one that we will adopt, the word 'theory' in 'type theory' refers to the type-theoretic formalism that we have set up, together with the stipulation that all inductive types exist. In particular, there is a variable collection of inference rules in the theory depending on which inductive types have been written down. This is similar to the situation in set theory, where the *axiom schema of replacement* likewise comprises an infinite number of axioms.

Booleans. A very simple inductive type is `Bool`, the type of Booleans. It can likewise be formed in any context, and there are two constructors not taking any arguments:

- $0 : \mathtt{Bool}$.
- $1 : \mathtt{Bool}$.

The induction principle (elimination rule) for `Bool` states that, in order to construct a dependent function in $\prod_{x:\mathtt{Bool}} C(x)$ for some $C : \mathtt{Bool} \to \mathcal{U}$, it suffices to specify elements $y : C(0)$ and $z : C(1)$, and this results in a function $\mathrm{ind}_{\mathtt{Bool}}(C, y, z) : \prod_{x:\mathtt{Bool}} C(x)$. The computation rules are

$$\mathrm{ind}_{\mathtt{Bool}}(C, y, z, 1) \equiv y, \qquad \mathrm{ind}_{\mathtt{Bool}}(C, y, z, 0) \equiv z.$$

Note that we use the symbol '0' in an ambiguous way: strictly speaking, we should not use the same symbol for $0 : \mathbb{N}$ and $0 : \mathtt{Bool}$. However, which one we mean should be clear from the context.

Binary trees. Another inductive type, important in computer programming applications, is `BinTree`, the type of *binary trees* (see Figure 1). There are two ways to construct a binary tree:

- $\mathtt{treeroot} : \mathtt{BinTree}$, the binary tree corresponding to only one node called "root",
- $\mathtt{join} : \mathtt{BinTree} \to \mathtt{BinTree} \to \mathtt{BinTree}$, a constructor taking two binary trees and appending them as the two branches of a new root node.

This is a nice example of an inductive type with a constructor taking more than one argument from the type itself. As an example, Figure 1 depicts the binary tree

$$\mathtt{join}\Big(\mathtt{join}(\mathtt{join}(\mathtt{treeroot}, \mathtt{treeroot}), \mathtt{treeroot}),$$
$$\mathtt{join}(\mathtt{join}(\mathtt{treeroot}, \mathtt{join}(\mathtt{treeroot}, \mathtt{treeroot})), \mathtt{join}(\mathtt{treeroot}, \mathtt{treeroot}))\Big).$$

We leave it to the reader to write down the elimination and computation rules for `BinTree`.

Equalities between inductive types

The types `Bool` and $\mathbf{1} + \mathbf{1}$ seem very similar: the first one is generated by two constructors without any arguments, while the second one is generated by $\mathrm{inl} : \mathbf{1} \to \mathbf{1} + \mathbf{1}$ and $\mathrm{inr} : \mathbf{1} \to \mathbf{1} + \mathbf{1}$, both of which essentially correspond to functions taking no arguments, since the unit type $\mathbf{1}$ has, on the intuitive level, only one element. So how do these two types relate?

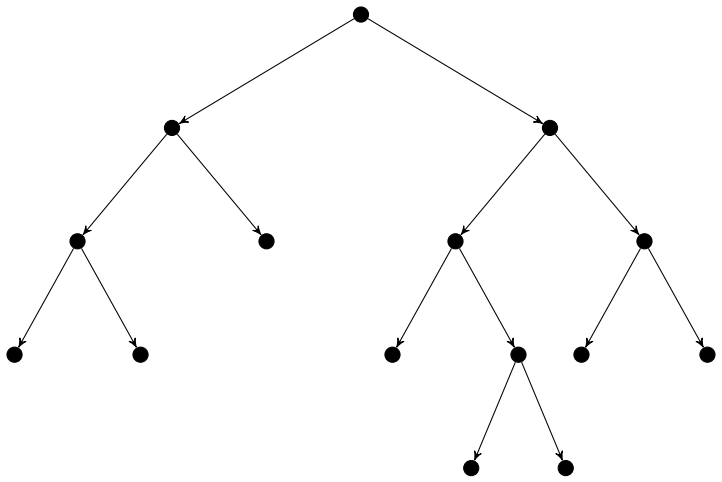

Figure 1: Example of a binary tree.

As a first step, we construct a map $f : \mathtt{Bool} \to \mathbf{1} + \mathbf{1}$. We explain its construction first informally, and then formally. One obtains a function out of $\mathtt{Bool}$ by specifying its values $f(0)$ and $f(1)$ and then applying the elimination rule. In this case, we can simply define $f(0) :\equiv \mathrm{inl}(\star)$ and $f(1) :\equiv \mathrm{inr}(\star)$.

Formally, we have the following proof tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{y : \mathtt{Bool}, x : \mathtt{Bool} \vdash \mathbf{1} + \mathbf{1} : \mathcal{U}} \quad \dfrac{\vdots}{y : \mathtt{Bool} \vdash \mathrm{inl}(\star) : \mathbf{1} + \mathbf{1}} \quad \dfrac{\vdots}{y : \mathtt{Bool}, \vdash \mathrm{inr}(\star) : \mathbf{1} + \mathbf{1}} \quad \dfrac{\vdots}{y : \mathtt{Bool} \vdash y : \mathtt{Bool}}}{y : \mathtt{Bool} \vdash \mathrm{ind}_{\mathtt{Bool}}(x.\mathbf{1} + \mathbf{1}, \mathrm{inl}(\star), \mathrm{inr}(\star), y) : \mathbf{1} + \mathbf{1}}}{\vdash \lambda(y : \mathtt{Bool}).\mathrm{ind}_{\mathtt{Bool}}(x.\mathbf{1} + \mathbf{1}, \mathrm{inl}(\star), \mathrm{inr}(\star), y) : \mathtt{Bool} \to \mathbf{1} + \mathbf{1}}$$

In terms of the functional form, we also could have written the desired function more concisely as $\mathrm{ind}'_{\mathtt{Bool}}(\lambda x.\mathbf{1} + \mathbf{1}, \mathrm{inl}(\star), \mathrm{inr}(\star)) \ : \ \mathtt{Bool} \to \mathbf{1} + \mathbf{1}$. Summing up, we constructed $f : \mathtt{Bool} \to \mathbf{1} + \mathbf{1}$, simply by mapping $0 \mapsto \mathrm{inl}(\star)$ and $1 \mapsto \mathrm{inr}(\star)$ and applying induction over $\mathtt{Bool}$.

Similarly, we can construct a function $g : \mathbf{1} + \mathbf{1} \to \mathtt{Bool}$. Informally, we define $g(z)$ for $z : \mathbf{1} + \mathbf{1}$ by induction on z, which reduces the problem to constructing $g(z)$ in the cases $z \equiv \mathrm{inl}(x)$ and $z \equiv \mathrm{inr}(y)$ for $x, y : \mathbf{1}$. In the former case, we put $g(\mathrm{inl}(x)) :\equiv 0$, while in the latter case, $g(\mathrm{inr}(y)) :\equiv 1$. Formally, this is expressed by the following proof tree:

$$\dfrac{\dfrac{\dfrac{\vdots}{x : \mathbf{1} + \mathbf{1}, p : \mathbf{1} + \mathbf{1} \vdash \mathtt{Bool} : \mathcal{U}} \quad \dfrac{\vdots}{x : \mathbf{1} + \mathbf{1}, a : \mathbf{1} \vdash 0 : \mathtt{Bool}} \quad \dfrac{\vdots}{x : \mathbf{1} + \mathbf{1}, b : \mathbf{1} \vdash 1 : \mathtt{Bool}} \quad \dfrac{\vdots}{x : \mathbf{1} + \mathbf{1} \vdash x : \mathbf{1} + \mathbf{1}}}{x : \mathbf{1} + \mathbf{1} \vdash \mathrm{ind}_{\mathbf{1}+\mathbf{1}}(p.\mathtt{Bool}, a.0, b.1, x) : \mathtt{Bool}}}{\vdash \lambda(x : \mathbf{1} + \mathbf{1}).\mathrm{ind}_{\mathbf{1}+\mathbf{1}}(p.\mathtt{Bool}, a.0, b.1, x) : \mathbf{1} + \mathbf{1} \to \mathtt{Bool}}$$

In terms of the functional form, we could also have written the desired function more concisely as $\mathrm{ind}'_{\mathbf{1}+\mathbf{1}}(\lambda p.\mathtt{Bool}, \lambda a.0, \lambda b.1) \ : \ \mathbf{1} + \mathbf{1} \to \mathtt{Bool}$.

We can now consider the compositions $g \circ f : \mathtt{Bool} \to \mathtt{Bool}$ and $f \circ g : \mathbf{1} + \mathbf{1} \to \mathbf{1} + \mathbf{1}$ and ask whether they are judgementally equal to the identity function $\lambda x.x$. For the first composition, we

obtain, for any $x : \mathtt{Bool}$,

$$(g \circ f)(x) \equiv g(f(x)) \equiv g(\mathrm{ind}_{\mathtt{Bool}}(\mathbf{1} + \mathbf{1}, \mathrm{inl}(\star), \mathrm{inr}(\star), x)).$$

However, already when trying to evaluate this further, we are stuck: we cannot apply any computation rule since neither x nor the argument of g is a 'base case' obtained by applying a constructor. Of course, intuitively we know that x corresponds to one of the base cases. But this is merely an intuitive restatement of the elimination rule, which lets us reduce the proof of a proposition to proving it in each base case. And the above judgemental equality is not a proposition, since it is not itself a type, hence the elimination rule is not applicable!

A similarly vexing problem arises for the other composition $f \circ g$. For any $x : \mathbf{1} + \mathbf{1}$, we have

$$(f \circ g)(x) \equiv f(g(x)) \equiv f(\mathrm{ind}_{\mathbf{1}+\mathbf{1}}(p.\mathtt{Bool}, 0, 1, x)),$$

and no computation rule applies in this situation.

There are many other situation with inductive types which 'ought to' be the same, but really are not. We will soon learn how to properly deal with this kind of problem using a notion of *propositional equality* together with the *univalence axiom*. These will indeed give rise to an equality between the types under consideration, and we will be able to prove the equation $\mathbf{1} + \mathbf{1} = \mathtt{Bool}$.

Inductive Type Families

Coproducts. Also the coproduct type $A + B$ can similarly be considered as an inductive type with two constructors:

- $\mathrm{inl} : A \to A + B$.
- $\mathrm{inr} : B \to A + B$.

Again, for any dependent type $C : A + B \to \mathcal{U}$, the elimination rule tells us that in order to construct an element of $C(x)$ for any given $x : A + B$, or equivalently a dependent function in $\prod_{x:A+B} C(x)$, it is enough to construct dependent functions $f : \prod_{a:A} C(\mathrm{inl}(a))$ and $g : \prod_{b:B} C(\mathrm{inr}(b))$, since these give rise to $\mathrm{ind}_{A+B}(C, f, g) : \prod_{x:A+B} C(x)$. The computation rule tells us that

$$\mathrm{ind}_{A+B}(C, f, g, \mathrm{inl}(a)) \equiv f(a), \qquad \mathrm{ind}_{A+B}(C, f, g, \mathrm{inr}(b)) \equiv g(b).$$

But actually, it is better not to consider $A + B$ as a single type: rather, it is a whole *family* of inductive types *parametrized* by A and B, corresponding to the fact that the formation rule for "+" involves A and B as types that need to be defined in the context under consideration. What this means is that we actually get a family of types $+ : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$, taking two types as input and outputting their coproduct.

Lists. Another example of an inductive type depending on another type as parameter is $\mathtt{List}(A)$, the type of all finite sequences (or lists) of elements of A. For this type, the constructors are:

- $\mathtt{nil} : \mathtt{List}(A)$ constructs the empty list,
- $\mathtt{cons} : A \to \mathtt{List}(A) \to \mathtt{List}(A)$, which maps an element $a : A$ and a list $L : \mathtt{List}(A)$ to the list $\mathtt{cons}(a, L) : \mathtt{List}(A)$, which corresponds to L together with a appended at the end. (Or appended at the beginning—this is a matter of interpretation.)

Strictly speaking, one should indicate the dependence on A in these constructors and write $\mathtt{nil}(A)$ and $\mathtt{cons}(A)$ instead of $\mathtt{nil}$ and $\mathtt{cons}$, respectively. But as long as one considers only a fixed A, no confusion can arise and there is no harm in omitting A as an explicit argument.

Lists of fixed length. There is another way in which an inductive type may depend on another type. To see why this might be the case, note that all the inductive types that we defined so far have the characteristic that the constructors have fixed codomain. But sometimes, it is desirable to define a whole family of inductive types *at once* by having constructors which take one or more arguments in some type(s) of this family, while returning an element of a potentially different type in the same family. In this case, we say that the type family is a dependent type *indexed* by the base type. Every parameter can also be considered to be an index, although in a trivial way.

As an example of this, we reconsider lists, but this time as indexed by their length: for any $n : \mathbb{N}$, there should be a type of lists $\mathtt{List}'(A, n)$ such that appending an element to a list should be considered a function $A \to \prod_{n:\mathbb{N}} \mathtt{List}'(A, n) \to \mathtt{List}'(A, \mathtt{succ}(n))$. In other words, $\mathtt{List}'(A)$ is a type family indexed by $\mathbb{N}$ and having constructors

- $\mathtt{nil}' : \mathtt{List}'(A, 0)$,
- $\mathtt{cons}' : A \to \prod_{n:\mathbb{N}} \mathtt{List}'(A, n) \to \mathtt{List}'(A, \mathtt{succ}(n))$.

In `Coq`, this looks as follows:

```
Inductive List' (A : Type) : nat → Type :=
| nil' : List' A 0
| cons' : A → forall n : nat, List' A n → List' A (S n).
```

More precisely, the functional forms of the inference rules are as follows: the formation rule is

$$\mathtt{List}' : \mathcal{U} \to \mathbb{N} \to \mathcal{U},$$

since $\mathtt{List}'$ produces a type $\mathtt{List}'(A, n)$ from any $A : \mathcal{U}$ and any $n : \mathbb{N}$. The introduction rules are (dependent) functions

$$\mathtt{nil}' : \prod_{A:\mathcal{U}} \mathtt{List}'(A, 0), \qquad \mathtt{cons}' : \prod_{A:\mathcal{U}} A \to \prod_{n:\mathbb{N}} \mathtt{List}'(A, n) \to \mathtt{List}'(A, \mathtt{succ}(n)).$$

For fixed A, the universal instance of the elimination rule is

$$\begin{aligned}\mathrm{ind}_{\mathtt{List}'(A)} : & \prod_{C:\prod_{n:\mathbb{N}} \mathtt{List}'(A,n) \to \mathcal{U}} C(0, \mathtt{nil}') \\ & \to \left(\prod_{m:\mathbb{N},\, x:A,\, u:\mathtt{List}'(A,m)} C(m,u) \to C(\mathtt{succ}(m), \mathtt{cons}'(x,m,u)) \right) \to \prod_{n:\mathbb{N}} \prod_{x:\mathtt{List}'(A,n)} C(n,x).\end{aligned}$$

Finally, the two computation rules state that

$$\mathrm{ind}_{\mathtt{List}'(A)}(C, b, s, 0, \mathtt{nil}'(A)) \equiv b,$$

$$\mathrm{ind}_{\mathtt{List}'(A)}(C, b, s, \mathtt{succ}(m), \mathtt{cons}'(x, m, u)) \equiv s(m, x, u, \mathrm{ind}_{\mathtt{List}'(A)}(C, b, s, m, u)).$$

The dependent type $\mathtt{List}'(A) : \mathbb{N} \to \mathcal{U}$ refines $\mathtt{List}(A) : \mathcal{U}$ in the sense that $\sum_{n:\mathbb{N}} \mathtt{List}'(A, n)$ obeys the same rules as $\mathtt{List}(A)$, but $\mathtt{List}'(A)$ contains more information in the sense that it also indexes every list by its length. In fact, it is straightforward to construct functions

$$\mathtt{List}(A) \to \sum_{n:\mathbb{N}} \mathtt{List}'(A, n), \qquad \left(\sum_{n:\mathbb{N}} \mathtt{List}'(A, n)\right) \to \mathtt{List}(A),$$

by applying elimination rules on the domain and the appropriate introduction rules on the codomain, and these functions 'ought to be' mutually inverse to each other, thereby defining an isomorphism between $\mathtt{List}(A)$ and $\sum_{n:\mathbb{N}} \mathtt{List}'(A, n)$. But again, the computation rules are not powerful enough to show that both compositions yield the identity, and therefore we will have to resort to the upcoming propositional equality and univalence axiom. This is exactly what we will start doing next time: our main example of an inductive type family are the so-called *identity types* which will give us a the new notion of propositional equality.

Example: a bit of arithmetic

In order to become more familiar with inductive types and induction principles, let us define the basic arithmetic of natural numbers. First of all, we would like to try and define a function

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

implementing addition of natural numbers. By induction over the first argument, it is sufficient to define $\mathtt{add}(0) : \mathbb{N} \to \mathbb{N}$, and $\mathtt{add}(\mathtt{succ}(n)) : \mathbb{N} \to \mathbb{N}$ in terms of $\mathtt{add}(n) : \mathbb{N} \to \mathbb{N}$. For the first, we simply take it to be the identity function,

$$\mathtt{add}(0) :\equiv \lambda n.n : \mathbb{N} \to \mathbb{N},$$

since adding 0 to any other number should reproduce that number. For the other function, we define

$$\mathtt{add}(\mathtt{succ}(n)) :\equiv \lambda m.\mathtt{succ}(\mathtt{add}(n, m)) : \mathbb{N} \to \mathbb{N},$$

since adding $\mathtt{succ}(n)$ to any other number should yield the successor of the sum of n and the other number. As an alternative (and not judgementally equal!) definition, we could have defined either or both of $\mathtt{add}(0)$ and $\mathtt{add}(\mathtt{succ}(n))$ also by a second induction over their remaining argument.

Defining multiplication can now be done similarly in terms of addition. We define

$$\mathtt{mult}(0) :\equiv \lambda n.0 : \mathbb{N} \to \mathbb{N},$$

and

$$\mathtt{mult}(\mathtt{succ}(n)) :\equiv \lambda m.\mathtt{add}(n, \mathtt{mult}(n, m)) : \mathbb{N} \to \mathbb{N},$$

and by induction, this yields a function $\mathtt{mult} : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$. Again, there are variants on this definition, but they all result in the same theory of arithmetic, which is precisely the usual one.