Induction on Derivations is Recursion

Jules Jacobs

April 15, 2021

Abstract

This note is an attempt to clarify something that used to confuse me. We'll see that induction on derivations is really recursion and not induction, which means that impredicative Church encodings for inductive relations actually work. These are the same as the Knaster-Tarski theorem.

When we define an inductive data type like natural numbers,

$$\mathbb{N} ::= 0 \mid S(\mathbb{N})$$

we can do proofs using induction,

$$P(0) \implies (\forall n \in \mathbb{N}, P(n) \implies P(S(n))) \implies (\forall n \in \mathbb{N}, P(n)),$$

and we can do recursion by making a function $\mathsf{rec}(z, f) : T \to T$ out of $z : T$ and $f : T \to T$,

$$\begin{aligned} \mathsf{rec}(f, z, 0) &= z \\ \mathsf{rec}(f, z, S(n)) &= f(\mathsf{rec}(f, z, n)) \end{aligned}$$

These two are closely related: the type of rec is

$$T \to (T \to T) \to T,$$

which matches the form of induction if we put $P(n) := T$. In type theory, where implication ($\implies$) and function space ($\to$) coincide, this is literally true. Recursion is the special case of induction where the type is constant. Or: induction is the generalisation of recursion where the type is allowed to depend on the input.

Given an inductive type like $\mathbb{N}$, recursion allows us to define a function

$$\mathbb{N} \to T$$

whereas induction allows us to define a dependent function

$$(n : \mathbb{N}) \to T(n)$$

which may be written $\forall n : \mathbb{N}, T(n)$ or $\Pi n : \mathbb{N}, T(n)$, which is different syntax for the same thing.

The question I want to explore in this note is:

When we do induction on derivations, are we doing recursion or induction?

The question makes sense, because in a proof assistant like Coq, we use the same construct `Inductive` to both define the type natural numbers, and to define an inductive relation.

Let us use the inductively defined relation $L = (\leq)$ to figure this out:

$$\frac{.}{(0,0) \in L}\ \text{L\_ZERO} \qquad \frac{(n,m) \in L}{(n,S(m)) \in L}\ \text{L\_SUCC1} \qquad \frac{(n,m) \in L}{(S(n),S(m)) \in L}\ \text{L\_SUCC2}$$

Induction on derivations tells us the following schema to prove $\forall (n,m) \in L, P(n,m)$:

$$\begin{aligned}
&P(0,0) \implies \\
&(\forall n \in \mathbb{N}, m \in \mathbb{N}, P(n,m) \implies P(n,S(m))) \implies \\
&(\forall n \in \mathbb{N}, m \in \mathbb{N}, P(n,m) \implies P(S(n),S(m))) \implies \\
&(\forall (n,m) \in L, P(n,m))
\end{aligned}$$

Your initial reaction may be:

This is clearly induction, because the predicate $P(n,m)$ depends on n,m.

While this may seem plausible, it is not correct. We are doing induction on the *derivation* of $(n,m) \in L$, not on (n,m) itself. The predicate $P(n,m)$ does not depend on the *derivation* of $(n,m) \in L$.

What is a derivation? A derivation is built up out applications of L_zero, L_succ1, L_succ2. For example, a derivation of $(1,2) \in L$ is L_succ2(L_succ1(L_zero)), but another derivation of the same fact is L_succ1(L_succ2(L_zero)). So we have multiple different derivations of the same fact $(1,2) \in L$.

The predicate $P(n,m)$ does not depend on the derivation of $(n,m) \in L$.

If it were induction, then the predicate would have been $P(n,m,D)$ where D is a derivation of $(n,m) \in L$. Therefore, we conclude:

Induction on derivations is really recursion on derivations.

The dependence on (n,m) arises because it's baked into the definition of L. When seen as a predicate, L has type $L : (\mathbb{N} \times \mathbb{N}) \to \mathsf{Type}$. Compare this with $\mathbb{N} : \mathsf{Type}$. Since $\mathbb{N}$ doesn't depend on anything, its recursion principle $T \to (T \to T) \to T$ is entirely non-dependent. That the recursion principle for $L(n,m)$ depends on (n,m) is because L itself depends on (n,m). The type of an *induction* principle for $D \in L(n,m)$ would depend on the derivation D as well.

Sure, but who cares?

I don't know, but there are cases where this difference makes a difference. We can use impredicative Church encodings to define the natural numbers $\mathbb{N} := (T : \mathsf{Type}) \to T \to (T \to T) \to T$ by their recursion principle. Unfortunately, such a definition does not work well in ordinary type theory, because this gives us no way to do induction on $\mathbb{N}$. We can do recursion just fine, because a Church natural number *is* its own recursor. But we can't do induction. However:

If all we're doing with inductive relations is recursion,
then we should be able to define them using Church encodings!

Indeed, we can, via Knaster & Tarski's fixed point theorem. To do so, we first define a function $F : (\mathbb{N} \times N) \to (\mathbb{N} \times \mathbb{N})$ corresponding to the derivation rules L_zero, L_succ1, L_succ2:

$$F(A) := \{(0,0)\} \cup \{(n,S(m)) \mid (n,m) \in A\} \cup \{(S(n),S(m)) \mid (n,m) \in A\}$$

This function is monotone, and therefore Knaster-Tarski gives us that its least fixed point is:

$$\mathsf{lfp}(F) = \{A \subseteq \mathbb{N} \times \mathbb{N} \mid F(A) \subseteq A\}$$

We can mimic this in Coq by using predicates $T \to \mathsf{Prop}$ as sets:

```
Definition set (T : Type) := T → Prop.
```

Using this notion, we define subsets and intersection:

```
Definition subset (A B : set T) := ∀ x, A x → B x.

Definition intersection (P : set (set T)) := λ x, ∀ A, P A → A x.
```

Then we define the Knaster-Tarski least fixed point of a function from sets to sets:

```
Definition lfp (F : set T → set T) := intersection (λ A, subset (F A) A).
```

Lastly, we define the F given above:

```
Definition F : set (nat * nat) → set (nat * nat) :=
  ∀ A, λ '(n,m), (n = 0 ∧ m = 0) ∨
          (∃ m', m = S m' ∧ A (n,m')) ∨
          (∃ n' m', n = S n' ∧ m = S m' ∧ A (n',m')).
```

And we take its least fixed point:

```
Definition leq' := lfp F.
```

Okay, you've defined `leq'`, but how we do know that it's actually the relation we want?

To show that `leq'` is actually the right relation, let us define the original relation inductively:

```
Inductive leq : (nat * nat) → Prop :=
  | leq_zero : leq (0,0)
  | leq_succ1 n m : leq (n,m) → leq (n, S m)
  | leq_succ n m : leq (n,m) → leq (S n, S m).
```

And prove that this is equivalent: $\texttt{leq}(n,m) \iff \texttt{leq}'(n,m)$:

```
Lemma leq_leq' n m :
  leq (n,m) ⟺ leq' (n,m).
Proof.
  unfold leq', lfp, intersection, subset, F. split.
  - induction 1; naive_solver.
  - intros H. apply (H leq). intros [].
    naive_solver (eauto using leq).
Qed.
```

I think we can conclude the following:

Inductive relations do not give us more power;
they can be simulated using Church encodings.

The following Coq file contains all these definitions:

https://julesjacobs.com/notes/inductionondervations/iod.v

It also contains a version of `leq` defined in Type instead of Prop. We print the corresponding induction principles that Coq generates:

```
Inductive leq : (nat * nat) → Prop :=
  | leq_zero : leq (0,0)
  | leq_succ1 n m : leq (n,m) → leq (n, S m)
  | leq_succ n m : leq (n,m) → leq (S n, S m).

Check leq_ind.

Inductive leq_type : (nat * nat) → Type :=
  | leq_zero' : leq_type (0,0)
  | leq_succ1' n m : leq_type (n,m) → leq_type (n, S m)
  | leq_succ' n m : leq_type (n,m) → leq_type (S n, S m).

Check leq_type_ind.
```

The former only gives us the recursion principle, but the latter *does* give a true **induction** principle. The predicate for `leq_type_ind` *is* allowed to depend on the derivation. Run the Coq file to see for yourself!