A Duality in Proof Structures: Intuition for Bridging Induction and Coinduction

Tadayoshi Kamegai

First Version : Nov 23, 2024
Last Update : Nov 30, 2024

Contents

1 Introduction

This note aims to provide an intuition between induction and coinduction, by revisiting the notion of induction, and attempting to connect the two notions through duality of least and greatest fix points (or stable subsets). We then exemplify this through an application under the context of coq using naturals and conaturals.

2 Definitions

Definition 2.1 (Monotone Function) *Let U be any set, $F : \mathcal{P}(U) \to \mathcal{P}(U)$. F is monotone if*

$$X \subseteq Y \implies F(X) \subseteq F(Y)$$

From here onwards assume every F is monotone unless explicitly stated otherwise.

Definition 2.2 *X is*

1. *F-closed if $F(X) \subseteq X$*
2. *F-consistent if $X \subseteq F(X)$*
3. *a fixed point if $X = F(X)$*

Theorem 2.3 *Given some F,*

1. *The least fixed point of $F := \mu F$ is the intersection of all F-closed sets*
2. *The greatest fixed point of $F := \nu F$ is the union of all F-consistent sets*

Proof. Corollary of the Knaster-Tarski Theorem.

Corollary 2.4 (Principle of Induction and Coinduction) *Given some F and X,*

1. *If X if F-closed, $\mu F \subseteq X$*
2. *If X is F-consistent, $X \subseteq \nu F$*

Proof. Corollary of Theorem 2.3. □

Consequently, to prove some proposition of μF, it suffices to prove the proposition for some F-closed set X (and similarly with F-consistent for νF).

Example 2.5 (Mathematical Induction)

Consider the function $F : \mathcal{P}(\mathbb{N}) \to \mathcal{P}(\mathbb{N})$ by :

$$F(X) = \{0\} \cup \{x + 1 : x \in X\}$$

Then (as $\mathbb{N}$ is F-closed) $\mu F = \mathbb{N}$, so to prove some property P is satisfied for all $\mathbb{N}$, it suffices to show that $S = \{n \in \mathbb{N} : P(n)\}$ is F-closed. Specifically, we want to show $F(S) \subseteq S$.

Equivalently,

$$\begin{aligned} & \{0\} \cup \{x + 1 : x \in S\} \subseteq S \\ \iff & 0 \in S \text{ and } (\forall x \in \mathbb{N},\ x \in S \implies x + 1 \in S) \\ \iff & P(0) \text{ and } (\forall x \in \mathbb{N},\ P(x) \implies P(x + 1)) \end{aligned} \tag{1}$$

So mathematical induction is an instance of the principle of induction.

3 Under Programming Contexts

3.1 Induction on $\mathbb{N}$

First consider an implementation for the naturals in Coq.

```
Inductive nat : Type :=
    | O : nat              (* Base case: zero *)
    | S : nat -> nat.      (* Successor: the next natural number *)
```

Then, our function from Example 2.5 provides an inductive hypothesis that can be used to prove properties about `nat` (which Coq will automatically generate). Equivalently, we have the following:

```
Section nat_ind.

Variable P : nat -> Prop.
Hypothesis P0 : P O.
Hypothesis Pn : forall n, P n -> P (S n).

Fixpoint nat_ind (n : nat) : P n :=
    match n with
        | O => P0
        | S n' => Pn n' (nat_ind n')
    end.

End nat_ind.
```

Remark 3.1

The above is the most natural form of induction for `nat`, with hypothesis for each constructor. When it is not clear whether certain constructors are monotone however, Coq will generate the most obvious inductive hypothesis.

For instance, consider an alternative representation for `nat` like the following:

```
Inductive case (A : Type) : Type :=
    | id : A -> case A.

Inductive nat : Type :=
    | O : nat
    | S : case nat -> nat.
```

Here, if we consider the function $F_1 : \mathcal{P}(\texttt{nat}) \to \mathcal{P}(\texttt{nat})$ by :

$$F_1(X) = \{\texttt{O}\} \cup \{\texttt{S}\ x : x \in \texttt{case nat}\}\ (= \texttt{nat})$$

We generate a (weak, but) sufficient induction scheme to prove a proposition for `nat`. That is,

$$\begin{aligned} &\forall\ x \in \texttt{nat}, P(x) \\ \iff &P(\texttt{O}) \text{ and } (\forall\ x \in \texttt{case nat},\ P(\texttt{S}\ x)) \end{aligned} \tag{2}$$

This is, an equivalent proposition to splitting x into their constructors. (This is what vanilla Coq will automatically generate).

We can strengthen the induction scheme by setting up a more intricate function $F_2 : \mathcal{P}(\texttt{nat}) \to \mathcal{P}(\texttt{nat})$:

$$F_2(X) = \{\texttt{O}\} \cup \{\texttt{S}\ x : x \in \texttt{case}\ X\}$$

Then, $\mu F = \texttt{nat}$, and setting $S = \{n \in \texttt{nat} : P(n)\}$,

$$\begin{aligned} &F_2(S) \subseteq S \\ \iff &\{\texttt{O}\} \cup \{\texttt{S}\ (\texttt{id}\ x) : x \in X\} \subseteq X \\ \iff &P(\texttt{O}) \text{ and } (\forall x \in \texttt{nat},\ x \in S \implies \texttt{S}\ (\texttt{id}\ x) \in S) \\ \iff &P(\texttt{O}) \text{ and } (\forall x \in \texttt{nat},\ P(x) \implies P(\texttt{S}\ (\texttt{id}\ x))) \end{aligned} \tag{3}$$

which generates an induction scheme resembling the canonical induction scheme for the naturals. Of course, we need to check that F_2 is monotone, but this is straightforward.

3.2 Induction for more general Types

Consider a slightly more general inductive object with one of its constructors looking like the following :

```
Inductive Foo : Type :=
    | base  : Foo
    | arg_1 : List Foo -> Bar -> Foo
    | arg_2 : List Foo -> Option Foo -> Foo.
```

The canonical function here would be $F_{\texttt{Foo}} : \mathcal{P}(\texttt{Foo}) \to \mathcal{P}(\texttt{Foo})$ by :

$$F_{\texttt{Foo}}(X) = \{\texttt{base}\} \cup \{\texttt{arg\_1}\ x\ y : x \in \texttt{List}\ X,\ y \in \texttt{Bar}\} \cup \{\texttt{arg\_2}\ x\ y : x \in \texttt{List}\ X,\ y \in \texttt{Option}\ X\}$$

Similar to before, $\mu F_{\texttt{Foo}} = \texttt{Foo}$, and setting $S = \{n \in \texttt{Foo} : P(n)\}$,

$$\begin{aligned} & F_{\texttt{Foo}}(S) \subseteq S \\ \iff & \{\texttt{base}\} \cup \{\texttt{arg\_1}\ x\ y : x \in \texttt{List}\ S,\ y \in \texttt{Bar}\} \cup \\ & \{\texttt{arg\_2}\ x\ y : x \in \texttt{List}\ S,\ y \in \texttt{Option}\ S\} \subseteq S \\ \iff & P(\texttt{base}) \text{ and } (\forall x \in \texttt{List}\ S,\ y \in \texttt{Bar},\ P(\texttt{arg\_1}\ x\ y)) \\ & \text{and } (\forall x \in \texttt{List}\ S,\ y \in \texttt{Option}\ S,\ P(\texttt{arg\_2}\ x\ y)) \end{aligned} \tag{4}$$

This is our induction scheme on the condition that $F_{\texttt{Foo}}$ is monotone, which is sufficiently ensured if `List` and `Option` are monotone (which they are), and composition of monotone functions is monotone.

We now have to consider elements of S that are a part of $\texttt{List}\ S$ and $\texttt{Option}\ S$. Consider the proposition :

$$\forall x \in \texttt{List}\ S,\ y \in \texttt{Option}\ S,\ P(\texttt{arg\_2}\ x\ y)$$

Noting that $\texttt{Option}\ S = \{\texttt{None}\}\ \cup\ \{\texttt{Some}\ x : x \in S\}$, we want to show

$$\begin{aligned} & \forall x \in \texttt{List}\ S,\ y \in (\{\texttt{None}\} \cup \{\texttt{Some}\ x : x \in S\}),\ P(\texttt{arg\_2}\ x\ y) \\ \iff & \forall x \in \texttt{List}\ S,\ z \in S,\ P(\texttt{arg\_2}\ x\ \texttt{None})\ \wedge\ P(\texttt{arg\_2}\ x\ (\texttt{Some}\ z)) \end{aligned} \tag{5}$$

Focusing on the left of the conjunction, to show $\forall x \in \texttt{List}\ S,\ P(\texttt{arg\_2}\ x\ \texttt{None})$, we introduce a new object :

```
Inductive Forall {A} (P : A -> Prop) : List A -> Prop :=
    | Forall_nil  : Forall P []
    | Forall_cons : forall (x : A) (l : List A),
                    P x -> Forall P l -> Forall P (x :: l)
```

Now,

$$\begin{aligned} & \forall x \in \texttt{List}\ S,\ P(\texttt{arg\_2}\ x\ \texttt{None}) \\ \iff & \forall x \in \texttt{List Foo}, \texttt{Forall}\ P\ x \implies P(\texttt{arg\_2}\ x\ \texttt{None}) \end{aligned} \tag{6}$$

This strategy can be utilized to bring out information about S to the hypothesis for the general object, which when completed for the whole gives the full induction scheme for `Foo`.

Lemma 3.2 (Induction scheme for `Foo`)

Given some $P : \texttt{Foo} \to \texttt{Prop}$,

$$\begin{aligned} & \forall x \in \texttt{Foo}, P\ x \\ \iff & P(\texttt{base})\ \wedge\ (\forall x \in \texttt{List Foo},\ y \in \texttt{Bar},\ \texttt{Forall}\ P\ x \implies P(\texttt{arg\_1}\ x\ y)) \\ & \wedge\ (\forall x \in \texttt{List Foo},\ y \in \texttt{Foo},\ \texttt{Forall}\ P\ x \implies P(\texttt{arg\_2}\ x\ y)\ \wedge\ P(\texttt{arg\_2}\ x\ \texttt{None})) \end{aligned} \tag{7}$$

Proof. Follows from the method explained above.

For a more general case, constructors look something like

```
Inductive Foo ... : Type :=
    ...
    | arg_n : ... -> F_1 Foo -> ... -> F_m Foo -> ...
    ... .
```

for some n and m, where we assume that $\texttt{F\_1}, \cdots, \texttt{F\_m}$ are all monotonic, and a similar strategy can be employed.

As another variant, consider objects where the constructors are functions that return that object.

```
Inductive Fnat : Type :=
    | FO : Fnat
    | FS : (nat -> Fnat) -> Fnat.
```

Intuitively, to create an `Fnat` of the form `FS f` requires specifying some function `f`, meaning any `Fnat` of the form `f` n should be made with "less steps". Formally, we take the function $F_{\texttt{Fnat}} : \mathcal{P}(\texttt{Fnat}) \to \mathcal{P}(\texttt{Fnat})$ by

$$F_{\texttt{Fnat}}(X) = \{\texttt{FO}\} \cup \{\texttt{FS}\ f\ :\ \forall n \in \texttt{nat},\ f\ n \in X\}$$

Taking $S = \{x \in \texttt{Fnat} : P\ x\}$,

$$\begin{aligned} & F_{\texttt{Fnat}}(S) \subseteq S \\ \iff & \{\texttt{FO}\} \cup \{\texttt{FS}\ f\ :\ \forall n \in \texttt{nat},\ f\ n \in S\} \subseteq S \\ \iff & \texttt{FO} \in S \text{ and } (\forall f \in (\texttt{nat} \to \texttt{Fnat}),\ (\forall n \in \texttt{nat}, f\ n \in S) \implies \texttt{FS}\ f \in S) \\ \iff & P(\texttt{FO}) \text{ and } (\forall f \in (\texttt{nat} \to \texttt{Fnat}),\ (\forall n \in \texttt{nat}, P(f\ n)) \implies P(\texttt{FS}\ f)) \end{aligned} \tag{8}$$

which is what coq also automatically generates.

3.3 Induction as Folds

Recall the definition for folds on lists :

```
Fixpoint foldl {A B : Type} (H0 : B) (IH : A -> B -> B) (l : list A)
    : B :=
    match l with
        | [] => H0
        | x :: xs => IH x (foldl H0 IH xs)
    end.
```

which by observation closely resembles induction for lists.

3.4 Coinduction on $\overline{\mathbb{N}}$

In coq, implementing the conaturals is similar to that of the naturals.

```
CoInductive cnat : Type :=
    | CO : cnat              (* Base case: zero *)
    | CS : cnat -> cnat.     (* Successor: the next natural number *)
```

From the definition of the constructors, we can define a corresponding function $F_{\texttt{cnat}} : \mathcal{P}(\texttt{cnat}) \to \mathcal{P}(\texttt{cnat})$ by

$$F_{\texttt{cnat}}(X) = \{\texttt{CO}\} \cup \{\texttt{CS}\ x\ :\ \forall x \in X\}$$

which resembles a function similar to that of `nat` (as the constructors are of the same form). Then, we take the greatest fixed point of $F_{\texttt{cnat}}$ as our object, such that $\nu F_{\texttt{cnat}} = \texttt{cnat}$.

Lemma 3.3 *There exists an element in $\nu F_{\texttt{cnat}}$ that is not in $\mu F_{\texttt{cnat}}$.*

Proof. We claim that $\omega = \texttt{CS}\ (\texttt{CS}\ \ldots) \in \texttt{cnat}$, which is the infinite application of `CS` onto itself. By the coinduction principle, this follows if $\{\omega\}$ is a F-consistent set. The latter claim is straightforward, as $F(\{\omega\}) = \{\omega, \texttt{CO}\}$.

Now, $\omega \notin \mu F_{\texttt{cnat}}$, as the least fixed point has no elements with an infinite number of applications of the same constructor. □

Intuitively, inductive objects are only made from a finite application of their constructors, whereas coinductive objects can be made with infinite applications of their constructors. This is because the set of elements made by a finite number of applications of their constructors (in F) is F-closed and is a subset of μF (so is equal), and the set of elements made by an infinite number of applications of their constructors is F-consistent and is a superset of νF (so is also equal).

3.5 Equality of Coinduction in coq

General equality in coq is defined by a single constructor,

```
Inductive eq {A : Type} (x : A) : A -> Prop :=
    | eq_refl : eq x x
```

that states that any `x` is equal to `x`. Therefore, to prove equality in coq we resort to showing syntactic equality by some means of unfolding definitions. Consider the following :

```
CoFixpoint C1 := CS (CS C1).
Definition C2 := CS C1.
```

While intuitively the infinite unfolding of both `C1` and `C2` should be ω, there is no method to get syntactic equality of the two by unfolding, as `C1` will always have an even number of `CS` applications on `C1`, while `C2` will have an odd amount. Therefore, we often show equality at the bisimulation level (to claim that the two are equal up to any finite observation), and this is straightforward after defining a suitable object that represents bisimulation for `cnat`. This contrasts inductive objects, as equality for two inductive objects are decidable (by simply tracing back the finite construction tree), whereas equality between two general coinductive objects are undecidable.