Inductive Reasoning in Coq

Maria João Frade

HASLab - INESC TEC
Departamento de Informática, Universidade do Minho

2021/2022

Roadmap

- **Inductive Definitions**
  - inductive types and its elimination mechanisms;
  - proof by induction; case analysis; general recursion;
  - relations as inductive types; logical connectives as inductive types.
- **Coq: Pragmatic Features**
  - notation, overloading and interpretation scopes;
  - implicit arguments; proof irrelevance;
  - the Coq library; searching the environment;
  - useful tactics and commands; combining tactics; automatic tactics.

Inductive Definitions

Induction

Induction is a basic notion in logic and set theory.

- When a set is defined inductively we understand it as being "built up from the bottom" by a set of basic constructors.
- Elements of such a set can be decomposed in "smaller elements" in a well-founded manner.
- This gives us principles of
  - *"proof by induction"* and
  - *"function definition by recursion"*.

Inductive types

We can define a new type I inductively by giving its *constructors* together with their types which must be of the form

$$\boxed{\tau_1 \to \ldots \to \tau_n \to I \quad \text{, with} \quad n \geq 0}$$

- Constructors (which are the *introduction rules* of the type I) give the canonical ways of constructing one element of the new type I.
- The type I defined is the smallest set (of objects) closed under its introduction rules.
- The inhabitants of type I are the objects that can be obtained by a finite number of applications of the type constructors.

Type I (under definition) can occur in any of the "domains" of its constructors. However, the occurrences of I in τ_i must be in *positive positions* in order to assure the well-foundedness of the datatype.

For instance, assuming that I does not occur in types A and B: $I \to B \to I$, $A \to (B \to I) \to I$ or $((I \to A) \to B) \to A \to I$ are valid types for a constructor of I, but $(I \to A) \to I$ or $((A \to I) \to B) \to A \to I$ are not.

Inductive types - examples

The inductive type $\mathbb{N}$: Set of natural numbers has two constructors

$$\begin{array}{l} 0 : \mathbb{N} \\ \mathsf{S} : \mathbb{N} \to \mathbb{N} \end{array}$$

A well-known example of a higher-order datatype is the type $\mathbb{O}$: Set of ordinal notations which has three constructors

$$\begin{array}{lcl} \text{Zero} & : & \mathbb{O} \\ \text{Succ} & : & \mathbb{O} \to \mathbb{O} \\ \text{Lim} & : & (\mathbb{N} \to \mathbb{O}) \to \mathbb{O} \end{array}$$

To program and reason about an inductive type we must have means to analyse its inhabitants.

The *elimination rules* for the inductive types express ways to use the objects of the inductive type in order to define objects of other types, and are associated to new computational rules.

Recursors

When an inductive type is defined in a type theory the theory should automatically generate a scheme for proof-by-induction and a scheme for primitive recursion.

- The inductive type comes equipped with a *recursor* that can be used to define functions and prove properties on that type.
- The recursor is a constant $\mathbf{R}_I$ that represents the **structural induction principle** for the elements of the inductive type I, and the computation rule associated to it defines a safe recursive scheme for programming.

For example, $\mathbf{R}_{\mathbb{N}}$, the recursor for $\mathbb{N}$, has the following typing rule:

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Type} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi\, x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathbf{R}_{\mathbb{N}}\,P\,a\,a' : \Pi\, n{:}\mathbb{N}.\, P\,n}$$

and its computation rules are

$$\begin{array}{lcl} \mathbf{R}_{\mathbb{N}}\,P\,a\,a'\,0 & \to & a \\ \mathbf{R}_{\mathbb{N}}\,P\,a\,a'\,(\mathsf{S}\,x) & \to & a'\,x\,(\mathbf{R}_{\mathbb{N}}\,P\,a\,a'\,x) \end{array}$$

Proof-by-induction scheme

The proof-by-induction scheme can be recovered by setting P to be of type $\mathbb{N} \to \mathsf{Prop}$.

Let $\mathsf{ind}_{\mathbb{N}} := \lambda P{:}\mathbb{N} \to \mathsf{Prop}.\, \mathbf{R}_{\mathbb{N}}\,P$ we obtain the following rule

$$\frac{\Gamma \vdash P : \mathbb{N} \to \mathsf{Prop} \quad \Gamma \vdash a : P\,0 \quad \Gamma \vdash a' : \Pi\, x{:}\mathbb{N}.\, P\,x \to P\,(\mathsf{S}\,x)}{\Gamma \vdash \mathsf{ind}_{\mathbb{N}}\,P\,a\,a' : \Pi\, n{:}\mathbb{N}.\, P\,n}$$

This is the well known structural induction principle over natural numbers. It allows to prove some universal property of natural numbers $(\forall n{:}\mathbb{N}.\, Pn)$ by induction on n.

Primitive recursion scheme

The primitive recursion scheme (allowing dependent types) can be recovered by setting $P : \mathbb{N} \to \mathsf{Set}$.

Let $\mathsf{rec}_{\mathbb{N}} := \lambda P{:}\mathbb{N}\to\mathsf{Set}.\, \mathbf{R}_{\mathbb{N}}\, P$ we obtain the following rule

$$\frac{\Gamma \vdash T : \mathbb{N}\to\mathsf{Set} \quad \Gamma \vdash a : T\,0 \quad \Gamma \vdash a' : \Pi\, x{:}\mathbb{N}.\, T\, x \to T\,(\mathsf{S}\, x)}{\Gamma \vdash \mathsf{rec}_{\mathbb{N}}\, T\, a\, a' : \Pi\, n{:}\mathbb{N}.\, T\, n}$$

We can define functions using the recursors.

For instance, a function that doubles a natural number can be defined as follows:

$$\mathsf{double} \equiv \mathsf{rec}_{\mathbb{N}}\,(\lambda n{:}\mathbb{N}.\,\mathbb{N})\,0\,(\lambda x{:}\mathbb{N}.\,\lambda y{:}\mathbb{N}.\,\mathsf{S}\,(\mathsf{S}\,y))$$

This approach gives safe way to express recursion without introducing non-normalizable objects.

However, codifying recursive functions in terms of elimination constants is quite far from the way we are used to program. Instead we usually use general recursion and case analysis.

Case analysis

Case analyses gives an elimination rule for inductive types.

For instance, $n : \mathbb{N}$ means that n was introduced using either 0 or S, so we may define an object match n with $\{0 \Rightarrow b_1 \mid (\mathsf{S}\, x) \Rightarrow b_2\}$ in another type σ depending on which constructor was used to introduce n.

A typing rule for this construction is

$$\frac{\Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash b_1 : \sigma \quad \Gamma, x : \mathbb{N} \vdash b_2 : \sigma}{\Gamma \vdash \mathsf{match}\ n\ \mathsf{with}\ \{0 \Rightarrow b_1 \mid (\mathsf{S}\, x) \Rightarrow b_2\} : \sigma}$$

and the associated computation rules are

$$\begin{array}{lcl} \mathsf{match}\ 0\ \mathsf{with}\ \{0 \Rightarrow b_1 \mid (\mathsf{S}\, x) \Rightarrow b_2\} & \to & b_1 \\ \mathsf{match}\ (\mathsf{S}\, e)\ \mathsf{with}\ \{0 \Rightarrow b_1 \mid (\mathsf{S}\, x) \Rightarrow b_2\} & \to & b_2\,[e/x] \end{array}$$

The case analysis rule is very useful but it does not give a mechanism to define recursive functions.

General recursion

Functional programming languages feature *general recursion*, allowing recursive functions to be defined by means of pattern-matching and a general fixpoint operator to encode recursive calls.

The typing rule for $\mathbb{N}$ fixpoint expressions is

$$\frac{\Gamma \vdash \mathbb{N}\to\theta : s \quad \Gamma, f : \mathbb{N}\to\theta \vdash e : \mathbb{N}\to\theta}{\Gamma \vdash (\mathsf{fix}\ f = e) : \mathbb{N}\to\theta}$$

and the associated computation rules are

$$(\mathsf{fix}\ f = e)\, n \quad \to \quad e[(\mathsf{fix}\ f = e)/f]\, n$$

Of course, this approach opens the door to the introduction of non-normalisable objects.

Using this, the function that doubles a natural number can be defined by

$$(\mathsf{fix}\ \mathsf{double} = \lambda n : \mathbb{N}.\ \mathsf{match}\ n\ \mathsf{with}\ \{0 \Rightarrow 0 \mid (\mathsf{S}\, x) \Rightarrow \mathsf{S}\,(\mathsf{S}\,(\mathsf{double}\, x))\})$$

About termination

- Checking convertibility between types may require computing with recursive functions. So, the combination of non-normalization with dependent types leads to undecidable type checking.
- To enforce decidability of type checking, proof assistants either require recursive functions to be encoded in terms of recursors or allow restricted forms of fixpoint expressions.
- A usual way to ensure termination of fixpoint expressions is to impose syntactical restrictions constraining all recursive calls to be applied to terms structurally smaller than the formal argument of the function.
- Another way to ensure termination is to accept a measure function that specifies how the argument "decreases" between recursive function calls.

Computation

Recall that typing judgments in Coq are of the form $E\,|\,\Gamma \vdash M : A$, where E is the global environment and Γ is the local context.

Computations are performed as series of *reductions*.

β-reduction for compute the value of a function for an argument:

$$(\lambda x\!:\!A.\, M)\, N \quad \to_\beta \quad M[N/x]$$

δ-reduction for unfolding definitions:

$$M \quad \to_\delta \quad N \qquad \text{if } (M := N) \in E\,|\,\Gamma$$

ι-reduction for primitive recursion rules, general recursion and case analysis

ζ-reduction for local definitions: $\texttt{let}\ x := N\ \texttt{in}\ M \quad \to_\zeta \quad M[N/x]$

Note that the conversion rule is

$$\frac{E\,|\,\Gamma \vdash M : A \qquad E\,|\,\Gamma \vdash B : s}{E\,|\,\Gamma \vdash M : B} \quad \text{if } A =_{\beta\iota\delta\zeta} B \text{ and } s \in \{\mathsf{Prop}, \mathsf{Set}, \mathsf{Type}\}$$

Natural numbers

```
Inductive nat :Set :=   O : nat
                      | S : nat -> nat.
```

The declaration of this inductive type introduces in the global environment not only the constructors O and S but also the eliminators: nat_rect, nat_ind and nat_rec

```
Check nat_rect.
nat_rect
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

```
Print nat_ind.
nat_ind = fun P : nat -> Prop => nat_rect P
     : forall P : nat -> Prop,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

```
Print nat_rec.
nat_rec = fun P : nat -> Set => nat_rect P
     : forall P : nat -> Set,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```


Vectors of length n over A.

```
Inductive vector (A : Type) : nat -> Type :=
  | Vnil : vector A 0
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n).
```

Remark the difference between the two parameters A and n:
– A is a general parameter, global to all the introduction rules,
– n is an index, which is instantiated differently in the introduction rules.
The type of constructor Vcons is a dependent function.

```
Variables b1 b2 :  B.
Check (Vcons _ b1 _ (Vcons _ b2 _ (Vnil _))).
Vcons B b1 1 (Vcons B b2 0 (Vnil B))  : vector B 2
```

```
Check vector_rect.
vector_rect
     : forall (A : Type) (P : forall n : nat, vector A n -> Type),
       P 0 (Vnil A) ->
       (forall (a : A) (n : nat) (v : vector A n),
        P n v -> P (S n) (Vcons A a n v)) ->
       forall (n : nat) (v : vector A n), P n v
```


Equality

In Coq, the propositional equality between two inhabitants a and b of the same type A, noted $a = b$, is introduced as a family of recursive predicates "to be equal to a", parameterized by both a and its type A. This family of types has only one introduction rule, which corresponds to reflexivity.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : (eq A x x).
```

The induction principle of eq is very close to the Leibniz's equality but not exactly the same.

```
Check eq_ind.
eq_ind : forall (A : Type) (x : A) (P : A -> Prop),
         P x -> forall y : A, x = y -> P y
```

Notice that the syntax "a = b" is an abbreviation for "eq a b", and that the parameter A is implicit, as it can be inferred from a.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : x = x.
```

Relations as inductive types

Some relations can also be introduced as an inductive family of propositions. For instance, the order $n \leq m$ on natural numbers is defined as follows in the standard library:

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : (le n n)
  | le_S : forall m : nat, (le n m) -> (le n (S m)).
```

- Notice that in this definition `n` is a general parameter, while the second argument of `le` is an index. This definition introduces the binary relation $n \leq m$ as the family of unary predicates "to be greater or equal than a given n", parameterized by n.
- The Coq system provides a syntactic convention, so that "`le x y`" can be written "`x <= y`".
- The introduction rules of this type can be seen as rules for proving that a given integer n is less or equal than another one. In fact, an object of type $n \leq m$ is nothing but a proof built up using the constructors `le_n` and `le_S`.

Logical connectives in Coq

In the Coq system, most logical connectives are represented as inductive types, except for $\Rightarrow$ and $\forall$ which are directly represented by $\rightarrow$ and Π-types, negation which is defined as the implication of the absurd and equivalence which is defined as the conjunction of two implications.

```
Definition not := fun A : Prop => A -> False.

Notation "~ A" := (not A) (at level 75, right associativity).
```

```
Inductive True : Prop :=  I : True.
```

```
Inductive False : Prop := .
```

```
Inductive and (A : Prop) (B : Prop) : Prop :=
  | conj : A -> B -> (and A B).

Notation "A /\ B" := (and A B) (at level 80, right associativity).
```


Logical connectives in Coq

```
Inductive or (A : Prop) (B : Prop) : Prop :=
  | or_introl : A -> (or A B)
  | or_intror : B -> (or A B).

Notation "A \/ B" := (or A B) (at level 85, right associativity).
```

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : forall x : A, P x -> ex P.

exists x:A, P  is an abbreviation of  ex A (fun x:A => P).
```

```
Definition iff (P Q:Prop) := (P -> Q) /\ (Q -> P).

Notation "P <-> Q" := (iff P Q) (at level 95, no associativity).
```

The constructors are the introduction rules.
The induction principle gives the elimination rules.
All the (constructive) logical rules are now derivable.

Coq: Pragmatic Features

Exercises

Load the file `lesson2a.v` in the Coq proof assistant. Run the examples and analyse them.

Notation, overloading and interpretation scopes

To simplify the input of expressions, the Coq system introduces symbolic abbreviations (called *notations*) denoting some term or term pattern.

- Some notations are overloaded.
- One can find the function hidden behind a notation by using the `Locate` command.

```
Notation "A /\ B" := (and A B).
Locate "*".
Locate "/\".
```

Moreover, the Coq system provides a notion of *interpretation scopes*, which define how notations are interpreted.

- Scopes may be opened and several scopes may be opened at a time.
- When a given notation has several interpretations, the most recently opened scope takes precedence.
- One can use the syntax *(term)%key* to bound the interpretation of *term* to the scope *key*.

Implicit arguments

Some typing information in terms are redundant.

A subterm can be replaced by symbol _ if it can be inferred from the other parts of the term during typing.

```
Definition comp : forall A B C:Set, (A->B) -> (B->C) -> A -> C
  := fun A B C f g x => g (f x).

Definition example (A:Set) (f:nat->A) := comp _ _ _ S f.
```

The implicit arguments mechanism makes possible to avoid _ in Coq expressions. The arguments that could be inferred are automatically determined and declared as implicit arguments when a function is defined.

```
Set Implicit Arguments.

Definition comp1 : forall A B C:Set, (A->B) -> (B->C) -> A -> C
  := fun A B C f g x => g (f x).

Definition example1 (A:Set) (f:nat->A) := comp1 S f.
```


Implicit arguments

A special syntax (using `@`) allows to refer to the constant without implicit arguments.

```
Check (@comp1 nat nat nat S S).
```

It is also possible to specify an explicit value for an implicit argument.

```
Check (comp1 (C:=nat) S).
```

The generation of implicit arguments can be disabled with

```
Unset Implicit Arguments.
```

Proof irrelevance

Let `P` be a proposition and `t` a term of type `P`.

The following commands are not equivalent:

```
Theorem name :  P.
Proof t.

Definition name :  P := t.
```

- A definition made with `Definition` or `Let` is transparent: its value `t` and type `P` are both visible for later use.
- A definition made with `Theorem`, `Lemma`, etc., is opaque: only the type `P` and the existence of the value `t` are made visible for later use.
- Transparent definition can be unfolded and can be subject to δ-reduction, while opaque definitions cannot.

The Coq library

Proof development often take advantage from the large base of definitions and facts found in the Coq library.

- *The initial library*: it contains elementary logical notions and datatypes. It constitutes the basic state of the system directly available when running Coq.
- *The standard library*: general-purpose libraries containing various developments of Coq axiomatizations about sets, lists, sorting, arithmetic, etc. This library comes with the system and its modules are directly accessible through the `Require Import` command.
- *Users' contributions*: user-provided libraries or developments are provided by Coq users' community. These libraries and developments are available for download.

Coq standard library

In the Coq system most usual datatypes are represented as inductive types and packages provide a variety of properties, functions, and theorems around these datatypes.

Some often used packages:

`Logic`	Classical logic and dependent equality
`Arith`	Basic Peano arithmetic
`ZArith`	Basic relative integer arithmetic
`Bool`	Booleans (basic functions and results)
`Lists`	Polymorphic lists and Streams
`Sets`	Sets (classical, constructive, finite, infinite, power set, etc.)
`FSets`	Specification and implementations of finite sets and finite maps
`QArith`	Axiomatization of rational numbers
`Reals`	Formalization of real numbers
`Relations`	Relations (definitions and basic results)
...	

Searching the environment

Some useful commands to find already existing proofs of facts in the environment.

- `Search` *ident* - displays the name and type of all theorems of the current context whose statement's conclusion has the form (*ident* t1 .. tn)
- `SearchPattern` *pattern* - displays the name and type of all theorems of the current context which matches the expression *pattern*.
- `SearchRewrite` *pattern* - displays the name and type of all theorems of the current context whose statement's conclusion is an equality of which one side matches the expression *pattern*.

Check the following commands:

```
Search le.
SearchPattern (_ + _ <= _ + _).
SearchRewrite (_ + (_ - _)).
SearchRewrite (?A + (_ - ?A)).
```

Basic tactics

- `intro`, `intros` – introduction rule for Π (several times).
- `apply` – elimination rule for Π.
- `assumption` – match conclusion with an hypothesis.
- `exact` – gives directly the exact proof term of the goal.
- `contradiction` – attempts to find in the current context a contradiction.

Tactics for first-order reasoning

Proposition (P)	Introduction	Elimination (H of type P)
$\bot$		`elim` H, `contradiction`
$\neg A$	`intro`	`apply` H
$A \wedge B$	`split`	`elim` H, `destruct` H `as [H1 H2]`
$A \Rightarrow B$	`intro`	`apply` H
$A \vee B$	`left`, `right`	`elim` H, `destruct` H `as [H1|H2]`
$\forall x : A.\, Q$	`intro`	`apply` H
$\exists x : A.\, Q$	`exists` *witness*	`elim` H, `destruct` H `as [x H1]`

Tactics for equational reasoning

- `rewrite` – rewrites a goal using an equality.
- `rewrite <-` – rewrites a goal using an equality in the reverse direction.
- `reflexivity` – reflexivity property for equality.
- `symmetry` – symmetry property for equality.
- `transitivity` – transitivity property for equality.
- `replace a with b` – replaces `a` by `b` while generating the subgoal `a=b`.
- `f_equal` – appliable to goals of the form $f\, a_1 \ldots a_n = f'\, a'_1 \ldots a'_n$.
- ...

Convertibility tactics

- `simpl`, `red`, `cbv`, `lazy`, `compute` – performs evaluation.
- `unfold` – applies the δ rule for a transparent constant.
- `pattern` – performs a beta-expansion on the goal.
- `change` – replaces the goal by a convertible one.
- ...

Tactics for inductive reasoning

- `elim` – to apply the corresponding induction principle.
- `induction` – performs induction on an identifier.
- `case`, `destruct` – performs case analysis.
- `constructor` – applies to a goal such that the head of its conclusion is an inductive constant.
- `discriminate` – discriminates objects built from different constructors.
- `injection` – applies the fact that constructors of inductive types are injections.
- `inversion` – given an inductive type instance, find all the necessary condition that must hold on the arguments of its constructors.
- ...

Other useful tactics and commands

- `clear` – removes an hypothesis from the environment.
- `generalize` – reintroduces an hypothesis into the goal.
- `cut`, `assert` – proves the goal through an intermediate result.
- `absurd` – applies `False` elimination.
- `contradict` – allows to manipulate negated hypothesis and goals.
- `refine` – allows to give an exact proof but still with some holes ("_").
- ...

- `Admitted` – aborts the current proof and replaces the statement by an axiom that can be used in later proofs.
- `Abort` – aborts the current proof without saving anything.

Combining tactics

The basic tactics can be combined into more powerful tactics using tactics combinators, also called *tacticals*.

- `t1 ; t2` – applies tactic `t1` to the current goal and then `t2` to each generated subgoal.
- `t1 || t2` – applies tactic `t1`; if it fails then applies `t2`.
- `t ; [ t1 | ...| tn]` – applies `t` and then `ti` to the i-th generated subgoals; there must be exactly n subgoals generated by `t`.
- `idtac` – does nothing.
- `try t` – applies `t` if it does not fail; otherwise does nothing.
- `repeat t` – repeats `t` as long as it does not fail.
- `solve t` – applies `t` only if it solves the current goal.
- ...

Automatic tactics

- `trivial` – tries those tactics that can solve the goal in one step.
- `auto` – tries a combination of tactics `intro`, `apply` and `assumption` using the theorems stored in a database as hints for this tactic.
- `eauto` – like `auto` but more powerful but also more time-consuming.
- `tauto` – useful to prove facts that are tautologies in intuitionistic PL.
- `intuition` – useful to prove facts that are tautologies in intuitionistic PL.
- `firstorder` – useful to prove facts that are tautologies in intuitionistic FOL.
- `lia` – a tactic for linear integer arithmetic
- `nia` – a tactic for non-linear integer arithmetic
- `lra` – tactic for linear (real or rational) arithmetic.
- `ring` – does proves of equality for expressions containing addition and multiplication.
- `field` – like `ring` but for a field structure (it also considers division).
- `subst` – replaces all the occurrences of a variable defined in the hypotheses.

Controlling automation

Several *hint databases* are defined in the Coq standard library. The actual content of a database is the collection of the hints declared to belong to this database in each of the various modules currently loaded.

- `Hint Resolve` – add theorems to the database of hints to be used by `auto` using `apply`.
- `Hint Rewrite` – add theorems to the database of hints to be used by `autorewrite`
- ...

Defined databases: `core`, `arith`, `zarith`, `bool`, `datatypes`, `sets`, `typeclass_instances`, `v62`.

One can optionally declare a hint database using the command `Create HintDb`. If a hint is added to an unknown database, it will be automatically created.

Exercises

Load the file `lesson2b.v` in the Coq proof assistant. Analyse the examples and solve the exercises proposed.