Logic and Proofs

CSE203

Benjamin Werner — Pierre-Yves Strub

École polytechnique

Lecture 3

Data types and arithmetic

October 19th 2022

"real" objects

We have seen how propositions can "talk about" objects:

```
P : nat -> Prop

A : Type
R : A -> A -> Prop

forall x y, R x y -> R y x
```

But we do no know yet how concrete objects are constructed:

- what is `nat` ?
- what are `0, 1, 2`…

From here the logic of Coq becomes more computer specific

How it is done in set theory

Only sets

Axiom: there exists an empty set   we call it $\varnothing$

Let us define $0 \equiv \varnothing$

Axiom: for any X, there exists the set of subsets of X

so $\{\varnothing\}$ exists;   we call it 1

Axiom of unordered pairs : $\{\varnothing\ ;\{\varnothing\}\}$ exists; we call it 2

$3 \equiv \{\varnothing; \{\varnothing\}; \{\varnothing\ ; \{\varnothing\}\}\} = \{0; 1; 2\}$

$4 \equiv \{\varnothing; \{\varnothing\}; \{\varnothing\ ; \{\varnothing\}; \{\varnothing; \{\varnothing\}; \{\varnothing\ ; \{\varnothing\}\}\}\} = \{0; 1; 2; 3\}$

etc…

😳 😱

Data-types in Coq

First example : a type **`color`** with only two elements: **`red`** and **`blue`**

```
Inductive color : Type :=
  red   |   blue.
```

alternative

```
Inductive color : Type :=
    red : color
|   blue : color.
```

This defines :

`color : Type`

`red : color` and **`blue : color`**

two *constructors*

"Inductive" means that there are no other values in **`color`**

or equivalently : **`color`** is the smallest type containing the two constructors.

Computing with inductive types

Idea: a value of type **`color`** is **`red`** or **`blue`**;
we can *check* whether it is red or blue

```
Definition inv (c : color) :=
  match c with
  | red => blue
  | blue => red
  end.
```

alternative syntax:

```
Definition inv :=
fun c : color =>
  match c with
   | red => blue
   | blue => red
  end.
```

`inv : color -> color`

```
Eval compute in (inv red).
                     = blue   : color
```

Computations and deductions

Objects are identified *modulo computation*

`(inv red)` and **`blue`** are the <u>same</u> object

```
Lemma inv_red : inv red = blue.
Proof.
simpl.
reflexivity.
Qed.
```

Goal

```
inv red = blue

   blue = blue
```

`simpl` performs all computations possible in the goal.

In this case, **`simpl`** is not necessary; **`reflexivity`** does the computations if needed

Reasoning about inductive types

```
inv red = blue
inv (inv red) = red          inv (inv blue) = blue
```

all by computation: ok

But: `forall x : color, inv (inv x) = x`

- not possible computation because **`x`** is a <u>variable</u>
- we need reasonning
- we need to use the fact that there is no other **`color`** than **`red`** and **`blue`**

```
x : color
============================
inv (inv x) = x
```

reason <u>by case</u> over **`x`**

A proof by case

```
Lemma inv_involutive : forall c, inv (inv c) = c.
Proof.
move => c.
case: c.
+ simpl.
  reflexivity.
+ reflexivity.
Qed
```

creates two subgoals:
`c ↤ red` and
`c ↤ blue`

`inv (inv red) = red`

`inv (inv blue) = blue`

both subgoals are
solved by computation

alternatives: `elim: c.` `elim c.`

A little more complicated example

How many constructors: any finite number (actually up to 256)

But constructors can also have arguments.

For instance, we want a type **`option_nat`** whose elements are either:

- a **`nat`** ("regular" case)
- "nothing" (corresponding to a form of exception)

idea : **`div : nat -> nat -> option_nat`**

```
Inductive option_nat :=
| None : option_nat
| Some : nat -> option_nat.
```

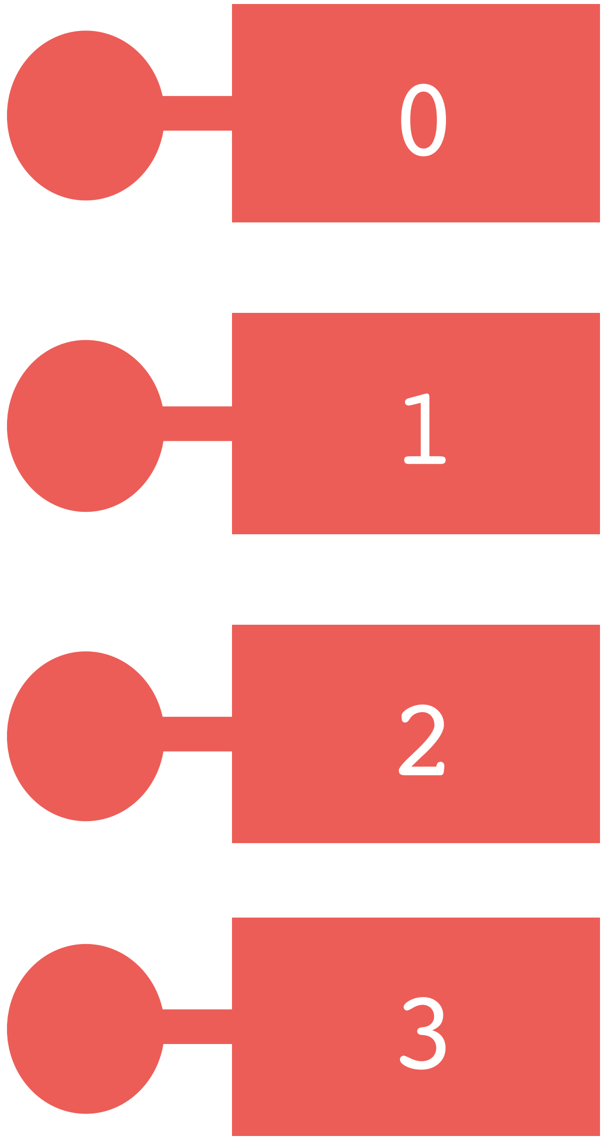
Possible values:
`None, (Some 0), (Some 1)…`

The values of option_nat

(Some 0)

(Some 1)

etc…



…

A little more complicated example (2)

translate `option_nat` into `nat` :

```
Definition convert n :=
  match n with
  | (Some a) => a
  | None => 0
  end.
```

```
Definition add_opt n m :=
  match n, m with
  | Some a, Some b => Some (a+b)
  | None, Some _ => None
  | Some _, None => None
  | None, None  =>  None
  end.
```

```
Definition add_opt n m :=
  match n, m with
  | Some a, Some b => Some (a+b)
  | _, _ => None
  end.
```

shortcut for

Recursive inductive definition

Let us see how nat is defined conceptually.

0, 1, 2, 3 … we cannot have infinitely many constructors

Proposal:

- 0 is a natural number
- If n is a natural number, then S(n) (or `(S n)`) is a natural number

The smallest set/type verifying these two clauses

0 is a natural number
S(0) is a natural number
S(S(0)) is a natural number
S(S(S(0))) is a natural number…

The smallest set is $\{S^n(0), n \in \mathbb{N}\}$

Recursive inductive type

In Coq

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

recursive argument

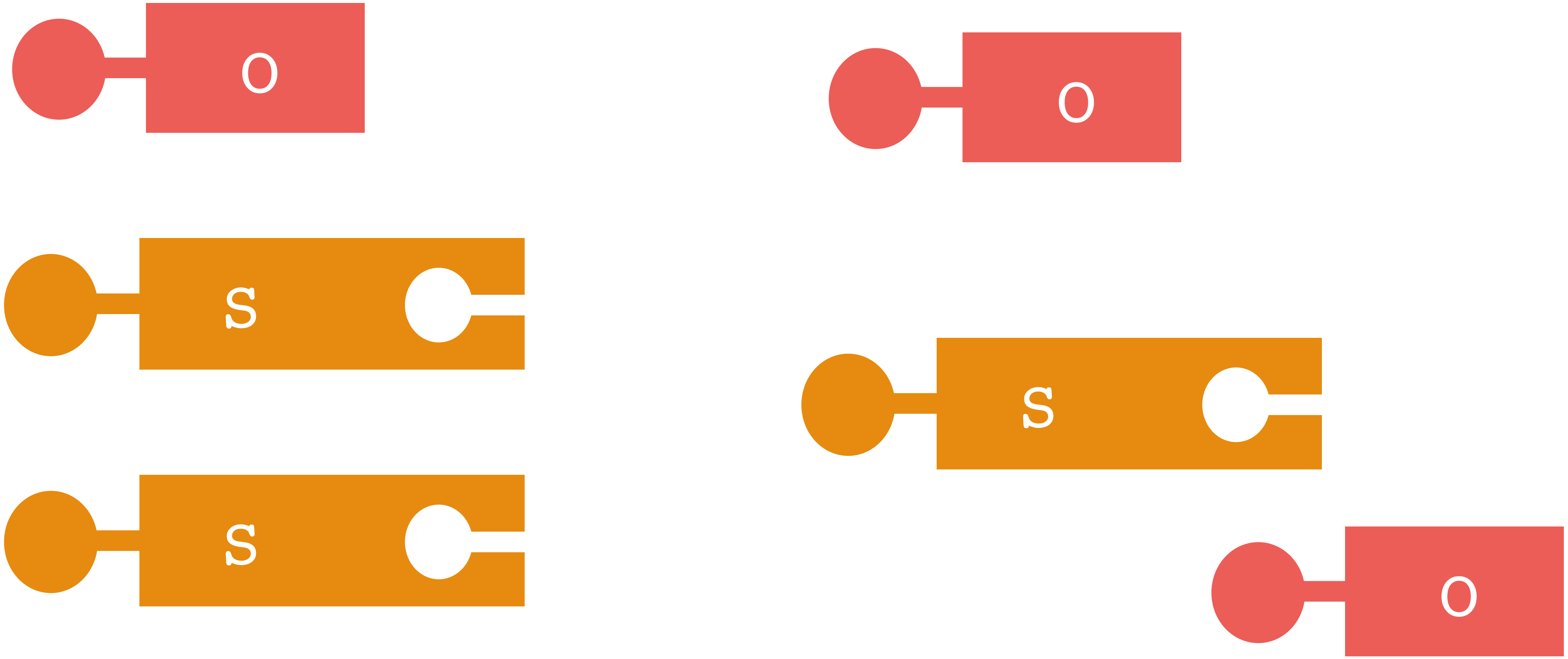
the smallest Type *closed* by the two clauses/constructors

Two consequences:

- Recursive functions
- Inductive reasoning

`3` is just pretty-printing for `S(S (S O))`
`0` is pretty-printing for `O`

How nats are constructed



etc… all nats are (S (S (S …. 0…)))

Functions over `nat`

like before:

```
Definition pred (n : nat) :=
  match n with
  | O => O
  | S m => m
  end.
```

(pred 5) ▻ 4

Recursive:

```
Fixpoint double (n : nat) :=
 match n with
  | O => O
  | S m => S (S (double m))
end.
```

recursive call

(double 5) ▻ 10

Detail

```
Fixpoint double (n : nat) :=
 match n with
 | O => O
 | S m => S (S (double m))
end.
```

```
 double (S (S O))   ▻   (S (S (double (S O))))

                    ▻   (S (S (S (S (double O)))))

                    ▻   (S (S (S (S O))))
```

The addition function

```
Fixpoint add n m : nat :=
  match n with
  | O => m
  | S p => S (add p m)
  end.
```

```
(add 5 4) ▻ 9
(add 4 5) ▻ 9
(add 0 x) ▻ x
(add 1 x) ▻ (S x)
```

BUT :

```
(add x 1) ▻ (add x 1)
```

What is the induction principle ?

nat is the smallest type obtained with O and S

O : nat

(S O) : nat

(S (S O)) : nat

…

And that is all !

Consider a property P, such that:

(P 0)

(P x) -> (P (S x))

Then all natural numbers satisfy P !

(because it is the smallest type closed by O and S)

Proofs by induction

```
n : nat
==============
P n
```

```
induction n.
```

2 goals

```
==============
P 0
```

base case

induction hypothesis

```
n : nat
IHn : P n
==============
P (S n)
```

n+1 case

variant: `elim` tactic

```
n : nat
==============
P n
```

```
elim: n.
```

2 goals

Like a case analysis but with an additional induction hypothesis

```
==============
P 0
```

base case

```
==================================
forall n : nat, P n -> P (S n)
```

n+1 case

induction hypothesis

`elim` with intro-patterns

```
n : nat
==============
P n
```

```
elim: n => [ | p hp]
```

```
==============
P 0
```

```
p : nat
hp : P p
==============
P (S p)
```

Same as **`induction`** but allows to chose the identifiers

`elim` on the goal

```
==============
forall n : nat, P n
```

means we work on the beginning of the goal

nothing here

```
elim => [ | p hp]
```

```
==============
P 0
```

```
p : nat
hp : P p
==============
P (S p)
```

Terminating functions

There are no infinitively looping functions in Coq

```
Fixpoint foo (n : nat) : nat :=   foo n.
```

is refused by the system

Reason : it would break the system (next time)

One allows recursive calls only on subterms of the argument
(or one of the arguments)

Proofs by induction

Example of a lemma proved by induction :

```
∀ x y, x + y = y + x
```

One will have to go through the following steps first :

- prove `∀ x, 0 + x = x + 0`
- prove `∀ x, x = x + 0`
- prove `∀ x y, x + (S y) = S (x + y)`