Coq in a Hurry

Yves Bertot

Coq in a Hurry

Yves Bertot

February 2010

These notes provide a quick introduction to the Coq system and show how it can be used to define logical concepts and functions and reason about them. It is designed as a tutorial, so that readers can quickly start their own experiments, learning only a few of the capabilities of the system. A much more comprehensive study is provided in [1], which also provides an extensive collection of exercises to train on.

1 Expressions and logical formulas

The Coq system provides a language in which one handles formulas, verifies that they are well-formed, and proves them. Formulas may also contain functions and limited forms of computations are provided for these functions.

1.1 Writing correct formulas

The first thing you need to know is how you can check whether a formula is well-formed. The command is called `Check`. Here are a few examples, which use a few of the basic objects and types of the system. Commands are always terminated by a period. In the following, text in `this style` is text that the user sends to the Coq system, while text in *`this style`* is the answer of the system.

```
Check True.
True : Prop

Check False.
False : Prop

Check 3.
3 : nat

Check (3+4).
3 + 4 : nat

Check (3=5).
3=5 : Prop
```

```
Check (3,4).
(3,4) : nat * nat

Check ((3=5)/\True).
3 = 5 /\ True : Prop

Check nat -> Prop.
nat -> Prop : Type

Check (3 <= 6).
3 <= 6 : Prop
```

The notation A :B is used for two purposes: the first is to indicate that the type of the expression A is the expression B, the second purpose which will appear later is to express that A is a proof of B. This notation can also be used in expressions we build, if we want to be explicit about the type an expression should have:

```
Check (3,3=5):nat*Prop.
(3,3=5):nat * Prop
      : nat * Prop
```

Among the formulas, some can be read as logical propositions (they have type `Prop`), others may be read as numbers (they have type `nat`), others may be read as elements of more complex data structures. You can also try to check badly formed formulas and in this case the Coq system returns an informative error statement.

Complex formulas can be constructed by combining propositions with logical connectives, or other expressions with addition, multiplication, the pairing construct, and so on. You can also construct a new function by using the keyword `fun`, which replaces the λ symbol of lambda calculus and similar theories. The following `Check` command contains the description of a function that takes a number `x` of type `nat` as input and returns the proposition that `x` is equal to `3`. The next `Check` command contains a logical proposition which can be read as *For every natural number x, either x is less than 3 or there exists a natural number y such that $x = y + 3$.*

```
Check (fun x:nat => x = 3).
fun x : nat => x = 3 : nat -> Prop

Check (forall x:nat, x < 3 \/ (exists y:nat, x = y + 3)).
forall x : nat, x < 3 \/ (exists y : nat, x = y + 3) : Prop
```

The following `Check` command illustrates the use of a let .. in construct, which makes it possible to give a temporary name to an expression. In this case the expression is a function that takes as input a number and returns a pair of numbers. In this example, we also see that the notation for applying a function a function `f` to a value, here `3`, is simply written by writing the function name `f` and the argument `3` side-by-side. Contrary to usual mathematical practice, parentheses around the argument are not mandatory.

```
Check (let f := fun x => (x * 3,x) in f 3).
let f := fun x : nat => (x * 3, x) in f 3 : nat * nat
```

Please note that some notations are *overloaded*. For instance, the `*` sign is used both to represent conventional multiplication on numbers and the cartesian product on types. One can find the function hidden behind a notation by using the `Locate` command.

```
Locate "_ <= _".
Notation             Scope
"x <= y" := le x y    : nat_scope
                        (default interpretation)
```

The conditions for terms to be well-formed have two origins: first, the syntax must be respected (parentheses must be balanced, binary operators must have two arguments, etc); second, expressions must respect a type discipline. The `Check` command not only checks that expressions are well-formed but it also gives the type of expressions. For instance we can use the `Check` command to verify progressively that some expressions are well-formed.

```
Check True.
True : Prop

Check False.
False : Prop

Check and.
and : Prop -> Prop -> Prop

Check (and True False).
True /\ False : Prop
```

In the last example, `and` is a function that expects an argument of type `Prop` and returns a function of type `Prop -> Prop`. It can therefore be applied to `True`, which has the right type. But the function we obtain expects another argument of type `Prop` and it can be applied to the argument `False`. The notation

$$a \texttt{ -> } b \texttt{ -> } c$$

actually stands for

$$a \texttt{ -> } (b \texttt{ -> } c)$$

and the notation $f\ a\ b$ actually stands for $(f\ a)\ b$.

The last example also shows that the notation `/\` is an infix notation for the function `and`. You can check this by using the `Locate` command.

1.2 Evaluating expressions

You can also force the Coq system to evaluate an expression. This actually means that some symbolic computation is performed on this formula, and there are several strategies to perform this symbolic computation. One strategy is called `compute`, here is an example of evaluation:

```
Eval compute in
  let f := fun x => (x * 3, x) in f 3.
= (9, 3) : nat * nat
```

After executing this command, the function `f` is not defined, it was just defined temporarily inside the expression that was evaluated. Here `f` can be used only inside the expression `f 3`.

Exercise on functions (exercise 2.5 from [1]) Write a function that takes five arguments and returns their sum, use `Check` to verify that your description is well-formed, use `Eval` to force its computation on a sample of values. *solutions to all exercises are given at the end of these notes.*

2 Programming in Coq

In the Coq system, programs are usually represented by *functions*. Simple programs can be executed in the Coq system itself, more complex Coq programs can be transformed into programs in more conventional languages and executed outside Coq.

2.1 Defining new constants

You can define a new constant by using the keyword `Definition`. Here is an example:

```
Definition example1 := fun x : nat => x*x+2*x+1.
```

An alternative, exactly equivalent, definition could be:

```
Definition example1 (x : nat) := x*x+2*x+1.
```

After executing one of these commands, the function can be used in other commands:

```
Check example1.
example1 : nat -> nat

Eval compute in example 1.
= 4 : nat
```

Sometimes, you may want to forget a definition that you just made. This is done using the `Reset` command, giving as argument the name you just defined. Moreover, if you want to see the definition of an object, you can use the `Print` command. When using a specialized environment for developing your proofs, like Coqide or Proof-general, you simply need to move the point of execution from one place of the document to another, using arrows that are provided at the top of the working window.

2.2 boolean conditional expressions

One of the basic pre-defined types of Coq is the type of *boolean values*. This type contains two values `true` and `false`. However, to use boolean expressions fully, one needs to load a library, called `Bool`, this brings the definitions of boolean conjunction, boolean disjunction, boolean negation, etc.

```
Require Import Bool.
```

Boolean values can be tested with an `if ...then ...else ...` construct, as in the following example:

```
Eval compute in if true then 3 else 5.
= 3 : nat
```

Knowing what functions are provided for this datatype can be done using the commands `Search` or `SearchAbout`. For instance, here is the result of the `Search` command.

```
Search bool.
eqb: bool -> bool -> bool
ifb: bool -> bool -> bool -> bool
true: bool
false: bool
andb: bool -> bool -> bool
orb: bool -> bool -> bool
implb: bool -> bool -> bool
xorb: bool -> bool -> bool
negb: bool -> bool
```

2.3 Computing with natural numbers

Another of the basic pre-defined types of Coq is the type of *natural numbers*. Again, to use this datatype fully, one needs to load a library called `Arith`.

```
Require Import Arith.
```

For natural numbers (numbers from 0 to infinity), there is another way to describe conditional computation. The idea is to express that every number satisfisies one of two cases: either it is 0, or it is the successor of another natural number `p`, and this is written `S p`. Thus, `1` is actually `S 0`, 2 is `S (S 0)` and so on.

Thus, there are two cases for a natural number, either it `0` or it is the successor of another number. Expressing the distinction between the two cases is written as in the following function, which returns a boolean value. This boolean value is `true` exactly when the argument is 0.

```
Definition is_zero (n:nat) :=
  match n with
    0 => true
  | S p => false
  end.
```

The expression `match ...with ...end` is called a *pattern-matching* construct. The value of the expression depends on the pattern that `n` satisfies: if `n` fits in the pattern `0`, then the value of the expression is `true`. If `n` fits in the pattern `S p`, then the value is false. In the pattern `S p`, the symbol `S` is used to describe the concept of *successor*, so intuitively, if `n` fits the pattern `S p`, this means that we detected that `n` is the successor of some number. The occurrence of `p` in this pattern introduces a local variable, which receives the value of which `n` is the successor, and `p` can be used in the right hand side of the `=>` arrow sign.

Pattern-matching is used frequently to compute on natural numbers. For instance, the Coq system contains a function named `pred`, which maps any successor number larger than 0 to its predecessor and which maps 0 to 0.

```
Print pred.
pred = fun n : nat =>
  match n with | 0 => n | S u => u end
     : nat -> nat

Argument scope is [nat_scope]
```

The definition of `pred` uses the pattern-matching construct to express that any number of the form `S u` will be mapped to `u` and any number of the form `0` will be mapped to itself.

Another feature that is used frequently to compute on natural numbers is *recursivity.* When defining a function using pattern matching, we are allowed to use the function that we are defining on the predecessor of the argument. Coq is very sensitive to recursive definitions, and we have to use a specific keyword to express that recursion will be used in the definition. This keyword is called `Fixpoint`. Here is an example of a recursive function that adds the first natural numbers:

```
Fixpoint sum_n n :=
  match n with
    0 => 0
  | S p => p + sum_n p
  end.
```

When describing a recursive function, we have to respect a constraint called *structural recursion.* The recursive call can only be made on a subterm of the initial argument. More precisely the argument of the recursive call must be a variable, and this variable must have been obtained from the initial argument through pattern matching. This constraint is used to ensure that every computation terminates, even if it involves recursive function. The definition of `sum_n` is well-formed, because `sum_n` is only used on `p`, which is a variable in the pattern `S p` used to match the initial argument `n`.

For instance, the following definition is not accepted because it fails to respect the contraint of *structural recursion.*

```
Fixpoint rec_bad n :=
  match n with 0 => 0 | S p => rec_bad (S p) end.
Error:
```

```
Recursive definition of rec_bad is ill-formed.
In environment
rec_bad : nat -> nat
n : nat
p : nat
Recursive call to rec_bad has principal argument equal to
"S p"
instead of p.
```

Recursive functions may have several arguments. In this case, the constraint of structural recursion must be satisfied by one of the argument positions. For instance, the following function takes an intermediate sum as argument and adds the first natural numbers to this extra argument. The constraint of structural recursion is satisfied for the first argument: in the recursive call, `p` is the argument in first position and it is obtained by pattern-matching from the initial argument in first position, `n`:

```
Fixpoint sum_n2 n s :=
 match n with
   0 => s
 | S p => sum_n2 p (p + s)
 end.
```

We can also define recursive functions that return values in other datatypes, like the type of boolean values, as in the following function that computes whether its argument is even:

```
Fixpoint evenb n :=
  match n with
    0 => true
  | 1 => false
  | S (S p) => evenb p
  end.
```

This function also shows new possibilities: we can use *deep pattern-matching* and have more than two cases in the pattern-matching construct. However, Coq will still check that all cases are covered, and recursive calls are still allowed only on variables obtained through pattern-matching.

2.4 Computing with lists

Several pieces of data of the same type can be grouped in lists. To use this, we need to tell Coq to load another library.

```
Require Import List.
```

To group several elements in the same list, we simply need to write them together, separated by two columns '`::`', and to add `'::nil'` at the end.

```
Check 1::2::3::nil.
1 :: 2 :: 3 :: nil
     : list nat
```

The notation '`::`' is actually used to add an element at the head of another list. Thus, `nil` represents an empty list, `3::nil` represents a list with only the element `3`, and `2::3::nil` represents a list with `2` on the head of a list with only the element `3`.

The `nil` expression has a special status in the `Coq` system, because it needs to know in which context it is used. In a list containing some elements, the `nil` expression is understood to represent an empty list for the type of these elements. However, the Coq system has trouble handling the `nil` expression when it is in complete isolation.

```
Check nil.
Error: Cannot infer the implicit parameter A of nil.
```

To circumvent this problem, we can tell Coq that the `nil` expression is to be used in the context where a specific list type is expected.

```
Check (nil : list nat).
nil:list nat
     : list nat
```

There are a few pre-defined functions on lists. For instance, there is a list concatenation function, named `app`, and usually noted with `++`, and there is a list processing function `map` that takes as inputs a function and list of values, and returns the list of results of this function applied to these values.

```
Eval compute in map (fun x => x + 3) (1::3::2::nil).
= 4::6::5::nil : list nat

Eval compute in map S (1::22::3::nil).
= 2::23::4::nil : list nat

Eval compute in
  let l := (1::2::3::nil) in l ++ map (fun x => x + 3) l.
= 1::2::3::4::5::6::nil : list nat
```

Exercise on lists, `map`, and `app` Define a function that takes as input a number n and returns a list with n elements, from 0 to $n-1$.

To access the elements of a list, we can use the same pattern-matching construct as the one we used for natural numbers, with a small change. Now, a list can satisfy one of two cases: either it is the empty list, or it has an element at the head of another list. In the second case, the pattern-matching construct makes it possible to give names to the first element and the rest of the list. See in the following example, which returns `true` if the argument list has at least one element and this element is even:

```
Definition head_evb l :=
  match l with nil => false | a::tl => evenb a end.
```

We can also use recursion on lists, with the same kind of structural recursion constraint as for recursion on natural numbers. Recursive calls can only be made on direct sublists. For instance, the following function adds up all the elements present in a list:

```
Fixpoint sum_list l :=
  match l with nil => 0 | n::tl => n + sum_list tl end.
```

As an example, we can write a simple program that sorts a list of natural numbers, knowing that the Coq system already provides a function called `leb` which compares two natural numbers and returns a boolean value.

```
Fixpoint insert n l :=
  match l with
    nil => n::nil
  | a::tl => if leb n a then n::l else a::insert n tl
  end.

Fixpoint sort l :=
  match l with
    nil => nil
  | a::tl => insert a (sort tl)
  end.
```

We can then test this function on a simple example:

```
Eval compute in sort (1::4::3::22::5::16::7::nil).
1::3::4::5::7::16::22::nil : list nat
```

Exercise on sorting Define a function that takes a list as input and returns `true` when it has less than 2 elements or when the first element is smaller than or equal to the second one. Then define a function that takes a list as input and returns `true` exactly when this list is sorted (Hint: when the list has at least two elements, the first element must be smaller than the second element and the tail must be sorted).

Exercise on counting Knowing that the Coq system provides a function `beq_nat` to compare two natural numbers, define a function `count_list` that takes a natural number and a list and returns the number of times the natural number occurs in the list.

3 Propositions and proofs

The notation A:B is actually used for several purposes in the Coq system. One of these purposes is to express that A is a proof for the logical formula B[1].

To construct proofs of formulas, we can simply look for existing proofs. Alternatively, we can construct new proofs and give them a name.

[1]This habit of viewing types as logical propositions is called the *Curry-Howard Isomorphism*

3.1 Finding existing proofs

One can find already existing proofs of facts by using the Search command. Its argument should always be an identifier.

```
Search True.
I : True

Search le.
between_le: forall (P : nat -> Prop) (k l : nat),
   between P k l -> k <= l
exists_le_S:
  forall (Q : nat -> Prop) (k l : nat),
      exists_between Q k l -> S k <= l
...
plus_le_reg_l: forall n m p : nat, p + n <= p + m -> n <= m
plus_le_compat_l: forall n m p : nat, n <= m -> p + n <= p + m
plus_le_compat_r: forall n m p : nat, n <= m -> n + p <= m + p
le_plus_l: forall n m : nat, n <= n + m
...
le_n : forall n : nat, n <= n
le_S : forall n m : nat, n <= m ->  n <= S m
```

The theorem le_S uses a function S, this function maps any natural number x to its successor $x+1$. Actually, the notation 3 is only a notation for S (S (S O)). The statement of le_S reads as : *for every n and m, numbers of type* nat*, if n is less than or equal to m, then n is less than or equal to the successor of m.* The arrow, ->, which was already used to represent function types is used here to represent implication between propositions. In practice, the ambiguity between the two uses of -> is never a problem.

The command SearchPattern takes a pattern as argument, where some of the arguments of a predicate can be replaced by incomplete expressions (we use a special anonymous variable _ to represent holes in incomplete expressions).

```
SearchPattern (_ + _ <= _ + _).
plus_le_compat_l: forall n m p: nat, n <= m -> p + n <= p + m
plus_le_compat_r: forall n m p: nat, n <= m -> n + p <= m + p
plus_le_compat: forall n m p q: nat, n <= m -> p <= q -> n + p <= m + q
```

The command SearchRewrite is similar, but it only looks for rewriting theorems, that is, theorems where the proved predicate is an equality. The listed theorems are those for which the pattern fits one of the members of the equality in the conclusion.

```
SearchRewrite (_ + (_ - _)).
le_plus_minus: forall n m : nat, n <= m -> m = n + (m - n)
le_plus_minus_r: forall n m : nat, n <= m -> n + (m - n) = m
```

The command SearchAbout makes it possible to find all theorems that are related to a given symbol. For instance, we used a function leb when describing a sorting program. We can look for all theorems related to this function.

```
SearchAbout leb.
leb_correct: forall m n : nat, m <= n -> leb m n = true
leb_complete: forall m n : nat, leb m n = true -> m <= n
leb_correct_conv: forall m n : nat, m < n -> leb n m = false
leb_complete_conv: forall m n : nat, leb n m = false -> m < n
leb_compare: forall n m : nat, leb n m = true <-> nat_compare n m <> Gt
```

3.2 Constructing new proofs

The usual approach to construct proofs is known as *goal directed proof*, with the following type of scenario:

1. the user enters a statement that he wants to prove, using the command `Theorem` or `Lemma`, at the same time giving a name for later reference,
2. the Coq system displays the formula as a formula to be proved, possibly giving a context of local facts that can be used for this proof (the context is displayed above a horizontal line written `=====`, the goal is displayed under the horizontal line),
3. the user enters a command to decompose the goal into simpler ones,
4. the Coq system displays a list of formulas that still need to be proved,
5. back to step 3.

The commands used at step 3 are called *tactics*. Some of these tactics actually decrease the number of goals. When there are no more goals the proof is complete, it needs to be saved, this is performed when the user sends the command `Qed.` The effect of this command is to save a new theorem whose name was given at step 1. Here is an example:

```
Lemma example2 : forall a b:Prop, a /\ b -> b /\ a.
1 subgoal

  ============================
   forall a b : Prop, a /\ b -> b /\ a

Proof.
 intros a b H.
1 subgoal

  a : Prop
  b : Prop
  H : a /\ b
  ============================
   b /\ a

split.
2 subgoals
...
```

```
  H : a /\ b
  ============================
   b

subgoal 2 is:
 a

destruct H as [H1 H2].
...
  H1 : a
  H2 : b
  ============================
   b

exact H2.
1 subgoal
...
  H : a /\ b
  ============================
   a

intuition.
Proof completed.

Qed.
intros a b H.
split.
 destruct H as [H1 H2]
   exact H2.
 intuition.
example2 is defined
```

The theorem that is defined is called `example2`, it can later be used in other tactics. This proof uses several steps to describe elementary reasoning steps. There is a large collection of tactics in the Coq system, each of which is adapted to a shape of goal. Each goal actually has two parts, a *context* and a *conclusion*. The elements of the context usually have the form `a : type` or `H : formula`, and we call them *hypotheses*. They represent facts that we can temporarily assume to hold.

The tactic `destruct H as [H1 H2]` was adapted because the hypothesis `H` was a proof of a conjunction of two propositions. The effect of the tactic was to add two new hypotheses in the context, named `H1` and `H2`. It is worthwhile remembering a collection of tactics for the basic logical connectives.

We list basic tactics in the following table, inspired from the table in [1] (p. 130). To use this table, you should remember that each connective can appear either in an hypothesis of the context (and we assume this hypothesis is named `H`) or in the conclusion of the goal. The tactic to be used in each case is not the same. When working on hypothesis, you should use the tactics listed in rows starting with "Hypothesis". For instance, the first step of the proof of `example2` works on a universal quantification, in the conclusion, hence the tactic that is used is `intros`. This `intros` tactic has three arguments `a`, `b`, and `H`, so it actually processes the first three connectives of the conclusion: two quantifications and an implication. The second step works on a conjunction `/\` and in the conclusion, hence the tactic is `split`. The third step works on a conjunction and in the hypothesis `H`, hence the tactic is `destruct H as [H1 H2]`. The fourth step is not referenced in the table: the tactic `exact H2` simply expresses that we want to prove a statement that is present in the context, in the hypothesis `H2`. A tactic `assumption` could also have been used to express that we want Coq to look for one hypothesis whose statement is the same as the conclusion. The fifth step performs a less elementary step: it calls an automatic tactic called `intuition`.

	$\Rightarrow$	$\forall$	$\wedge$

Hypothesis `H`	`apply H`	`apply H`	`elim H`
			`case H`
			`destruct H as [H1 H2]`

conclusion	`intros H`	`intros H`	`split`
	$\neg$	$\exists$	$\vee$

Hypothesis `H`	`elim H`
	`case H`
		`elim H`
		`case H`
		`destruct H as [x H1]`
			`elim H`
			`case H`
			`destruct H as [H1 | H2]`

conclusion	`intros H`	`exists` v	`left` or `right`
	$=$	`False`	

Hypothesis `H`	`rewrite H`
	`rewrite <- H`
		`elim H`
		`case H`	

conclusion	`reflexivity` `ring`		

When using the tactic `elim` or `case`, this usually creates new facts that are placed in the conclusion of resulting goals as premises of newly created implications. These premises must then be introduced in the context using the `intros` tactic. A quicker tactic does the two operations at once, this tactic is called `destruct`.

All tactics that work on hypotheses and take hypothesis names as arguments, like `destruct`, `apply`, or `rewrite`, can also take theorem names as arguments. Thus, every theorem you prove and save can later be used in other proofs.

3.3 One more example using basic tactics

We can illustrate these tactics on the proof of yet another example:

```
Lemma example3 : forall A B, A \/ B -> B \/ A.
```

The purpose of this proof is to show that the "or" logical connective is commutative. In English, the statement can be read as "for any two propositions `A` and `B`, if `A \/ B` holds (read this as `A` or `B`), then `B \/ A` holds". Our first reasoning step is to introduce the universally quantified propositions and the hypothesis in the context.

```
intros A B H.
...
A : Prop
B : Prop
H : A \/ B
  ============================
   B \/ A
```

Intuitively, this reasoning step corresponds to the sentence "let's fix two propositions `A` and `B` and let's assume that `A \/ B` holds, now we only have to prove `B \/ A`." After this, we need to think a little about how to attach the problem. To prove `A \/ B`, we could choose to work on the conclusion. This would correspond to choosing to prove either `A` or `B` (using the tactics `left` and `right`). However, if we only know that `A \/ B` holds, we cannot prove `A`, because `A \/ B` could be a consequence of the fact that `B` holds, but not `A`. Thus, working on the conclusion directly does not lead to a feasible proof. On the other hand, we can choose to work on the hypothesis. If `A \/ B` holds, there actually are two cases: either `A` holds or `B` holds, and we can ask to see these two cases independently. This is done by using the tactic `destruct`.

```
destruct H as [H1 | H1].
...
H1 : A
  ============================
   B \/ A

Subgoal 2 is
   B \/ A
```

Two goals are generated, and the hypothesis `H1` in the first goal tells us that we are in the case where we know that `A` holds. We are now in a situation where we can work directly on the conclusion, choosing to prove the right-hand part `A`, because we know that in this case, it holds.

```
right; assumption.
...
H1 : B
  ============================
   B \/ A
```

The semi-column '`;`' is used to compose several tactics together. More precisely, the tactic on the right hand side is applied to all the goals produced by the tacti on the left hand side. Here, the tactic `right` transforms the goal's conclusion `B \/ A` into `A` (it states that we want to prove the right-hand part of the 'or' statement); then the tactic `assumption` proves this by finding an assumption with `A` as statement, namely `H1`. Because the tactic `right; assumption` solves the first goal, the Coq system answers by displaying only the second goal, which is now the only one. We can solve this goal in a similar manner by choosing to prove the left-hand side.

```
left; assumption.
Proof completed.
Qed.
```

As usual we finish our proof by writing `Qed`. This saves the proof as a theorem that we can use later.

3.4 Examples using `apply`

According to the table page 13, the tactic `apply` is especially adapted to work on hypotheses or theorems whose main connective is a universal quantification or an implication. It actually works by separating the conclusion of a theorem from its premises, by matching the conclusion of the theorem with the conclusion of the current goal to guess as many universally quantified variables as possible, and by generating a new goal for each premise.

More precisely, if a theorem `th` as a statement of the form:

```
forall x1 ... xk, A1 x1 ... xk -> A2 x1 ... xk -> ... ->  An x1 ... xk ->
   C x1 ... xk
```

and the current goal's conclusion has the shape

```
  =====================
   C a1 ... ak
```

Then the tactic `apply th` replaces the current goal with `n` goals, whose statements are `A1 a1 ... ak`, ... `An a1 ... ak`. This works well when all the variables that universally quantified (here `x1` ... `xn`) appear in the goal conclusion. By matching the theorem's conclusion with the goal conclusion, the Coq system is able to infer that `x1` has to be `a1`, `x2` has to be `a2`, and so on.

When a variable does not occur in the theorem's conclusion, the Coq system is not able to infer the value this variable must take, but the user can specify its value using a directive `with`.

Let's illustrate this with two examples. In the first example, we only illustrate the simple use of `apply`. We use two theorems about comparisons between natural numbers.

```
Check le_n.
le_n : forall n : nat, n <= n
Check le_S.
le_S : forall n m : nat, n <= m -> n <= S m

Lemma example4 : 3 <= 5.
```

We need to remember that `3` is actually a notation for `S (S (S 0))` and `5` is a notation for `S (S (S (S (S 0))))`. Now, in a first step, we can apply the theorem `le_S`. This will match the theorem's conclusion `n <= S m` with the goal `3 <= 5`, thus finding the right choice for the variables `n` and `m` is `3` and `4` respectively. A new goal, corresponding to the theorem's premise is created.

```
apply le_S.
  ============================
   3 <= 4
```

In a second step, we can apply again the same theorem. This time the Coq system figures out that `n` has to be `3` and `m` has to be `3`.

```
apply le_S.
  ============================
   3 <= 3
```

Now we get a goal that is an instance of `le_n`. This is again checked by the `apply` tactic, but this time there is no new suboal generated and we can save the proof.

```
apply le_n.
Proof completed.
Qed.
```

In a second example, let's illustrate the case where some variable that is universally quantified in the hypothesis or theorem statement does not appear in the theorem's conclusion. A typical theorem that exhibits this characteristic is a transitivity theorem, like the transitivity theorem for the order "less than or equal to" on natural numbers.

```
Check le_trans.
le_trans : forall n m p : nat, n <= m -> m <= p -> n <= p

Lemma example5 : forall x y, x <= 10 -> 10 <= y -> x <= y.
intros x y x10 y10.
1 subgoal

  x : nat
  y : nat
  x10 : x <= 10
  y10 : 10 <= y
  ============================
   x <= y
```

If we try to use `le_trans` with the `apply` tactic as before, we get an error message.

```
apply le_trans.
Error: Unable to find an instance for the variable m.
```

The variable `m` does not occur in the conclusion of the theorem. Matching the conclusions of the theorem and the goals does not help finding the value for this variable. The user can give the value by hand:

```
apply le_trans with (m := 10).
2 subgoals
  ============================
   x <= 10
```

```
subgoal 2 is:
 10 <= y
assumption.
assumption.
Qed.
```

3.4.1 Examples using `rewrite`

Equality is one of the most common way to express properties of programs. As a consequence, many theorems have a conclusion that is an equality. The most practical tactic to use these theorem is `rewrite`.

Most theorems are universally quantified and the values of the quantified variables must be guessed when the theorems are used. The `rewrite` guesses the values of the quantified variables by finding patterns of the left-hand side in the goal's conclusion and instanciating the variables accordingly. Let's illustrate this on an example.

```
Lemma example6 : forall x y, (x + y) * (x + y) = x*x + 2*x*y + y*y.
intros x y.
```

At this point we would like to use a theorem expressing distributivity, let's search for this theorem using the `SearchRewrite` command.

```
SearchRewrite (_ * (_ + _)).
mult_plus_distr_l: forall n m p : nat, n * (m + p) = n * m + n * p
```

We can tell the Coq system to rewrite with this theorem.

```
rewrite mult_plus_distr_l.
  ============================
   (x + y) * x + (x + y) * y = x * x + 2 * x * y + y * y
```

We can observe that the Coq found on its own that the universally quantified variables had to be instanciated in the following manner: `n` had to be `(x + y)`, `m` had to be `x`, and `p` had to be `y`. We can now use a similar distributivity theorem where multiplication is on the right-hand side to progress in our proof.

```
SearchRewrite ((_ + _) * _).
mult_plus_distr_r: forall n m p : nat, (n + m) * p = n * p + m * p

rewrite mult_plus_distr_r.
  ============================
   x * x + y * x + (x + y) * y = x * x + 2 * x * y + y * y
```

In the last step, there were two possible choices of instanciation, one where `p` had to be `x` and one where `p` had to be `y`. The Coq system chose the former, but we could have been more directive and have written `rewrite mult_plus_distr_r with (p:=x)`. In the next steps, we look for theorems that make it possible to reshape the left-hand side of the equality, and then we work on the right-hand side.

```
rewrite mult_plus_distr_r.
  ============================
   x * x + y * x + (x * y + y * y) = x * x + 2 * x * y + y * y

SearchRewrite (_ + (_ + _)).
plus_assoc: forall n m p : nat, n + (m + p) = n + m + p
plus_permute: forall n m p : nat, n + (m + p) = m + (n + p)
plus_assoc_reverse: forall n m p : nat, n + m + p = n + (m + p)
plus_permute_2_in_4:
   forall n m p q : nat, n + m + (p + q) = n + p + (m + q)

rewrite plus_assoc.
  ============================
   x * x + y * x + x * y + y * y = x * x + 2 * x * y + y * y
```

At this point, we would like to rewrite with a theorem from right to left, this is possible using the `<-` modifier.

```
rewrite <- plus_assoc with (n := x * x).
  ============================
   x * x + (y * x + x * y) + y * y = x * x + 2 * x * y + y * y
```

Now we want to use commutativity for multiplication. We can find the relevant theorem using the `SearchPattern` command.

```
SearchPattern (?x * ?y = ?y * ?x).
mult_comm: forall n m : nat, n * m = m * n

rewrite mult_comm with (n:= y) (m:=x).
  ============================
   x * x + (x * y + x * y) + y * y = x * x + 2 * x * y + y * y
```

At this point, we want to proceed step-by-step to show that `x * y + x * y` is the same as 2 `* x * y`, and we know that 2 is a notation for `S 1`. We look for rewriting theorems with the relevant shape.

```
SearchRewrite (S _ * _).
mult_1_l: forall n : nat, 1 * n = n
mult_succ_l: forall n m : nat, S n * m = n * m + m
...
```

In our goal, we have two occurrences of the expression `(x * y)` and we want to rewrite only one of them with `mult_1_l` from right to left. This is possible, using a tactic called `pattern` to limit the place where rewriting occurs, stating that we want only the first occurrence to be rewritten.

```
pattern (x * y) at 1; rewrite <- mult_1_l.
  ============================
   x * x + (1 * (x * y) + x * y) + y * y = x * x + 2 * x * y + y * y
```

```
rewrite <- mult_succ_l.
  ============================
   x * x + 2 * (x * y) + y * y = x * x + 2 * x * y + y * y
```

At this point, we only need to use the associativity of multiplication.

```
SearchRewrite (_ * ( _ * _)).
mult_assoc_reverse: forall n m p : nat, n * m * p = n * (m * p)
mult_assoc: forall n m p : nat, n * (m * p) = n * m * p
rewrite mult_assoc.
  ============================
   x * x + 2 * x * y + y * y = x * x + 2 * x * y + y * y
```

We can now conclude using `reflexivity` as prescribed by the table page 13.

```
reflexivity.
Proof completed.
Qed.
```

Sometimes rewriting theorems have premises stating that the equality is valid only under some condition. When rewriting is performed with using theorems, the corresponding instance of premises is added as extra goals to be proved later.

3.5 More advanced tactics

A common approach to proving difficult propositions is to assert intermediary steps using a tactic called `assert`. Given an argument of the form `(H : `P`)`, this tactic yields two goals, where the first one is to prove P, while the second one is to prove the same statement as before, but in a context where a hypothesis named `H` and with the statement P is added.

Some automatic tactics are also provided for a variety of purposes, `intuition` and `tauto` are often useful to prove facts that are tautologies in propositional logic (try it whenever the proof only involves manipulations of conjunction, disjunction, and negation, for instance `example2` could have been solved by only one call to `intuition`); `firstorder` can be used for tautologies in first-order logic (try it when ever the proof only involves the same connectives, plus existential and universal quantification) `auto` is an extensible tactic that tries to apply a collection of theorems that were provided beforehand by the user, `eauto` is like `auto`, it is more powerful but also more time-consuming, `ring` mostly does proofs of equality for expressions containing addition and multiplication, `omega` proves systems of linear inequations. To use `omega` you first need to require that Coq loads it into memory. An inequation is linear when it is of the form $A \leq B$ or $A < B$, and A and B are sums of terms of the form $n * x$, where x is a variable and n is a numeric constant, 1, 3, -15, etcetera). The tactic `omega` can also be used for problems, which are *instances* of systems of linear inequations. Here is an example, where `f x` is not a linear term, but the whole system of inequations is the instance of a linear system.

```
Require Import Omega.
```

```
Lemma omega_example :
  forall f x y, 0 < x -> 0 < f x -> 3 * f x <= 2 * y -> f x <= y.
intros; omega.
Qed.
```

At this point, you should be able to perform the following two exercises.

Exercise on logical connectives (Exercise 5.6 from [1]) Prove the following theorems:

```
forall A B C:Prop, A/\(B/\C)->(A/\B)/\C
forall A B C D: Prop,(A->B)/\(C->D)/\A/\C -> B/\D
forall A: Prop, ~(A/\~A)
forall A B C: Prop, A\/(B\/C)->(A\/B)\/C
forall A B: Prop, (A\/B)/\~A -> B
```

Two benefits can be taken from this exercise. In a first step you should try using only the basic tactics given in the table page 13. In a second step, you can verify which of these statements are directly solved by the tactic `intuition`.

Exercise on universal quantification Prove

```
forall A:Type, forall P Q: A->Prop,
    (forall x, P x)\/(forall y, Q y)->forall x, P x\/Q x.
```

4 Proving properties of programs on numbers

To prove properties of functions computing on natural numbers, one can often show that several computations are related in some way. When the function being studied is recursive, this usually involves a proof by induction. The tactic one needs to know are:

- `induction` will make it possible to perform a proof by induction on a natural number n. This decompose the proof in two parts: first we need to verify that the property to prove is satisfy when n is 0, second we need to prove the property for `S` p is a consequence of the same property for p (the hypothesis about the property for p is called the *induction hypothesis*, its name usually starts with `IH`). Two goals are created, one for 0 and one for `S` p.
- `simpl` will replace calls of a recursive function on a complex argument by the corresponding value, as stated by the definition of the function.
- `discriminate` will be usable when one hypothesis asserts that `0 = S ...` or `true = false`.
- `injection` will be usable when one hypothesis has the form `S` x `=` `S` y to deduce x `=` y.

Let us look at an example proof. We want to show that the sum of the n first natural numbers satisfies a well-known polynomial formula.

```
Lemma sum_n_p : forall n, 2 * sum_n n + n = n * n.
induction n.
  ============================
   2 * sum_n 0 + 0 = 0 * 0

subgoal 2 is:
 2 * sum_n (S n) + S n = S n * S n
```

In the two generated goals, the first one has `n` replaced by `0`, the second one has `n` replaced by `S n` with a hidden hypothesis corresponding the statement being already true for `n`.

The first goal is very easy, because the definitions of `sum_n`, addition and multiplication make that both members of the equality are `0`. The tactic `reflexivity` solves this goal.

```
reflexivity.
```

After we send this tactic the second goal remains alone and is repeated in full. We can now see the induction hypothesis.

```
  n : nat
  IHn : 2 * sum_n n + n = n * n
  ============================
   2 * sum_n (S n) + S n = S n * S n
```

In this goal, we need to think a little. We can expand `S n * S n` into a formula where `n * n` occurs, and then rewrite with `IHn`, from right to left. We use the `assert` tactic to describe a intermediary equality between `S n * S n` the new value we want to use instead and the `ring` tactic to show that this replacement is valid.

```
assert (SnSn : S n * S n = n * n + 2 * n + 1).
  ============================
   S n * S n = n * n + 2 * n + 1

 ring.
  n : nat
  IHn : 2 * sum_n n + n = n * n
  SnSn : S n * S n = n * n + 2 * n + 1
  ============================
   2 * sum_n (S n) + S n = n * n + 2 * n + 1

rewrite SnSn.
  n : nat
  IHn : 2 * sum_n n + n = n * n
  SnSn : S n * S n = n * n + 2 * n + 1
  ============================
   2 * sum_n (S n) + S n = n * n + 2 * n + 1
```

We attack this goal by rewriting with the induction hypothesis, this replaces occurrences of the induction hypothesis right-hand side with by the left-hand side.

```
rewrite <- IHn.
  ============================
   2 * sum_n (S n) + S n = 2 * sum_n + n + 2 * n + 1
```

The next step is to make `sum_n (S n)` compute symbolically, so that its value is expressed usin `sum_n n` according to the definition. The tactic we use also forces the symbolic computation of multiplications.

```
simpl.
  ============================
   n + sum_n n + (n + sum_n n + 0) + S n =
   sum_n + (sum_n n + 0) + (n + (n + 0)) + 1
```

The next step is to make the `ring` tactic recognize that this equation is a consequence of associativity and commutativity of addition.

```
ring.
Qed.
```

When the recursive function has several levels of pattern matching, it is worthwhile proving statements about several successive numbers, as in the following proof about the function `evenb`.

```
Lemma evenb_p : forall n, evenb n = true -> exists x, n = 2 * x.
assert (Main: forall n, (evenb n = true -> exists x, n = 2 * x) /\
                  (evenb (S n) = true -> exists x, S n = 2 * x)).
```

At this point, we choose to prove a stronger statement, stating that the property we wish to prove is satisfied by both `n` and `S n`. We then proceed by a regular induction. Proving stronger statements by induction is a very common feature in proofs about recursive functions.

```
induction n.
  ============================
   (evenb 0 = true -> exists x : nat, 0 = 2 * x) /\
   (evenb 1 = true -> exists x : nat, 1 = 2 * x)

subgoal 2 is:
 (evenb (S n) = true -> exists x : nat, S n = 2 * x) /\
 (evenb (S (S n)) = true -> exists x : nat, S (S n) = 2 * x)
subgoal 3 is:
 forall n : nat, evenb n = true -> exists x : nat, n = 2 * x
```

The first goal has two parts, and for the first part, we need to work on an existential statement that occurs in the conclusion. Refering to the table of tactic on page 13, we see that we need to provide the existing value, using the `exists` tactic. Here the existing value for `x` is 0, since `0 = 2 * 0` is easy to prove. The second part is different: it is not possible to choose a value for `x`, but the hypothesis `evenb 1 = true` is inconsistent, because `evenb 1` actually computes to `false` and the equality `false = true` is not possible. When such an inconsistent case is reached we can use the `discriminate` tactic.

```
split.
  exists 0; ring.
simpl; intros H; discriminate H.
```

We do not detail the rest of the proof, but we give it for readers to try on their own computer.

```
split.
  destruct IHn as [_ IHn']; exact IHn'.
simpl; intros H; destruct IHn as [IHn' _].
assert (H' : exists x, n = 2 * x).
  apply IHn'; exact H.
destruct H' as [x q]; exists (x + 1); rewrite q; ring.
```

Once the Main hypothesis is proved, we can come back to our initial statement. At this point, the tactic firstorder is strong enough to finish the proof, but for didactic purposes, we will show another approach.

```
  Main : forall n : nat,
       (evenb n = true -> exists x : nat, n = 2 * x) /\
       (evenb (S n) = true -> exists x : nat, S n = 2 * x)
  ============================
   forall n : nat, evenb n = true -> exists x : nat, n = 2 * x
intros n ev.
...
  n : nat
  ev : evenb n = true
  ============================
   exists x : nat, n = 2 * x
```

Here we can use a facility of the Coq system that we haven't mentioned yet: when the statement of a theorem is a universal quantification or an implication, we can use this theorem as if it was a function: when applied to an expression it gives a new theorem whose statement is instantiated on this expression. In our case, we can use the hypothesis Main and instantiate it on n.

```
destruct (Main n) as [H _]; apply H; exact ev.
Qed.
```

Exercise on addition, alternative definition We can define a new addition function on natural numbers:

```
Fixpoint add n m := match n with 0 => m | S p => add p (S m) end.
```

Prove the following statements.

```
forall n m, add n (S m) = S (add n m)
forall n m, add (S n) m = S (add n m)
forall n m, add n m = n + m
```

Remember that you may use a lemma you just proved when proving a new exercise.

Exercise on the sum of odd numbers The sum of the first n odd natural numbers is defined with the following function:

```
Fixpoint sum_odd_n (n:nat) : nat :=
  match n with  0 => 0 | S p => 1 + 2 * p + sum_odd_n p end.
```

Prove the following statement:

```
forall n:nat, sum_odd_n n = n*n
```

5 Proving properties of programs on lists

When performing proofs on recursive functions that act on lists, we can also use proofs by induction, this time taking a list as the induction argument. Again, there will be two goals, the first one concerning the empty list, and the second one considering an arbitrary list of the form `a::l`, with an induction hypothesis on `l`.

Let us look at an example proof. We want to show that sorting a list with the `insert` and `sort` functions given earlier does not change the number of occurrences of each number in the list. Counting the number of occurrences of a number in a list is done with the following function:

```
Fixpoint count n l :=
  match l with
    nil => 0
  | a::tl =>
    let r := count n tl in if beq_nat n a then 1+r else r
  end.
```

We first want to show that inserting an element in a list always increases the number of occurrences for this number.

```
Lemma insert_incr : forall n l, count n (insert n l) = 1 + count n l.
intros n l; induction l.
  ============================
   count n (insert n nil) = 1 + count n nil
```

Here we use `simpl` to make Coq compute the various functions.

```
  ============================
   (if beq_nat n n then 1 else 0) = 1
```

Here we need to find existing theorems on `beq_nat`.

```
SearchAbout beq_nat.
beq_nat_refl: forall n : nat, true = beq_nat n n
beq_nat_eq: forall x y : nat, true = beq_nat x y -> x = y
beq_nat_true: forall x y : nat, beq_nat x y = true -> x = y
beq_nat_false: forall x y : nat, beq_nat x y = false -> x <> y
```

The existing theorem `beq_nat_refl` is useful for us, we use it with a `rewrite` tactic.

```
rewrite <- beq_nat_refl.
  ============================
   1 = 1
```

This goal is easily solved by `reflexivity`. The next goal is the step case of the proof by induction on lists.

```
reflexivity.
...
  IHl : count n (insert n l) = 1 + count n l
  ============================
   count n (insert n (a :: l)) = 1 + count n (a :: l)
```

Again, we first ask Coq to compute the various functions.

```
simpl.
  ============================
   count n (if leb n a then n :: a :: l else a :: insert n l) =
   S (if beq_nat n a then S (count n l) else count n l)
```

We first need to reason by cases on the expression `leb n a` and then by cases on the expression `beq_nat n a`. We use the tactic `case (leb n a)` for the first reasoning step. It produces two goals, one where `leb n a` is replaced by `true`, and one where it is replaced by `false`.

```
case (leb n a).
...
  ============================
   count n (n :: a :: l) =
   S (if beq_nat n a then S (count n l) else count n l)

subgoal 2 is:
 count n (a :: insert n l) =
 S (if beq_nat n a then S (count n l) else count n l)
```

We can now simplify the expression. We see that `beq_nat` appears again and we can re-use the theorem `beq_nat_refl`

```
simpl.
...
  ============================
   (if beq_nat n n
    then S (if beq_nat n a then S (count n l) else count n l)
    else if beq_nat n a then S (count n l) else count n l) =
   S (if beq_nat n a then S (count n l) else count n l)

rewrite <- beq_nat_refl.
```

```
...
  ============================
   S (if beq_nat n a then S (count n l) else count n l) =
   S (if beq_nat n a then S (count n l) else count n l)

reflexivity.
  IHl : count n (insert n l) = 1 + count n l
  ============================
   count n (a :: insert n l) =
   S (if beq_nat n a then S (count n l) else count n l)
```

In the last goal, we first simplify the expression.

```
simpl.
  IHl : count n (insert n l) = 1 + count n l
  ============================
   (if beq_nat n a then S (count n (insert n l))
    else count n (insert n l)) =
   S (if beq_nat n a then S (count n l) else count n l)
```

We now have to reason by cases on the value of `beq_nat n a`:

```
case (beq_nat n a).
...
  IHl : count n (insert n l) = 1 + count n l
  ============================
   S (count n (insert n l)) = S (S (count n l))

subgoal 2 is:
 count n (insert n l) = S (count n l)
```

In the two goals that are produced, rewriting with the hypothesis `IHl` will obviously lead to equalities where both sides are the same. We conclude the proof with the following two commands.

```
rewrite IHl; reflexivity.
rewrite IHl; reflexivity.
Qed.
```

6 Defining new datatypes

Numbers, boolean values, and lists are all examples of datatypes. When programming, we often need to define new datatypes, for example to express that the data can be fit a certain collection of cases, where each case contains a certain number of fields. Coq provides simple facilities to express this kind of data organization.

6.1 Defining inductive types

Here is an example of an inductive type definition:

```
Inductive bin : Type :=
  L : bin
| N : bin -> bin -> bin.
```

This defines a new type `bin`, whose type is `Type`, and expresses that the data in this type respects two cases. In the first case, there is no field and the data is noted `L`, in the second case there are two fields of type `bin` and the data can be written as `N` t_1 t_2. In other words, elements of the type `bin` can be obtained in two different ways, either by taking the constant `L` or by applying `N` to two objects of type `bin`.

```
Check N L (N L L).
N L (N L L) : bin
```

The type `bin` contains binary trees with no extra information carried in the internal nodes or leaves.

Exercise on datatype definition Define a datatype where there are three cases: a constant, a case where there are three fields, where the first field is a number and the next two fields are in the datatype being defined, and a case with four fields, where the first field is a boolean value and the three other fields are in the datatype being defined.

6.2 Pattern matching

Elements of inductive types can be processed using functions that perform some pattern-matching. For instance, we can write a function that returns the boolean value `false` when its argument is `N L L` and returns `true` otherwise.

```
Definition example7 (t : bin): bool :=
  match t with N L L => false | _ => true end.
```

6.3 Recursive function definition

We can define recursive functions on any inductive datatype using the `Fixpoint` command, but we have to respect the same structural recursion discipline as for natural numbers and lists: recursive calls can only be done on variables, and these variables must have been obtained from the initial argument through pattern-matching. Here are a few examples of recursive functions on our new datatype.

```
Fixpoint flatten_aux (t1 t2:bin) : bin :=
  match t1 with
    L => N L t2
  | N t'1 t'2 => flatten_aux t'1 (flatten_aux t'2 t2)
  end.
```

```
Fixpoint flatten (t:bin) : bin :=
  match t with
    L => L | N t1 t2 => flatten_aux t1 (flatten t2)
  end.

Fixpoint size (t:bin) : nat :=
  match t with
    L => 1 | N t1 t2 => 1 + size t1 + size t2
  end.
```

Structural recursive functions on user-defined data-types can be computed in a similar way to functions defined on numbers and lists.

```
Eval compute in flatten_aux (N L L) L.
 = N L (N L L) : bin
```

6.4 Proof by cases

Now that we have defined functions on our inductive type, we can prove properties of these functions. Here is a first example, where we perform a few case analyses on the elements of the type `bin`.

```
Lemma example7_size :
   forall t, example7 t = false -> size t = 3.
intros t; destruct t.
2 subgoals

  ============================
   example7 L = false -> size L = 3

subgoal 2 is:
 example7 (N t1 t2) = false -> size (N t1 t2) = 3
```

The tactic `destruct t` actually observes the various possible cases for `t` according to the inductive type definition. The term `t` can only be either obtained by `L`, or obtained by `N` applied to two other trees `t1` and `t2`. This is the reason why there are two subgoals.

We know the value that `example7` and `size` should take for the tree `L`. We can direct the Coq system to compute it:

```
simpl.
2 subgoals

  ============================
   true = false -> 1 = 3
```

After computation, we discover that assuming that the tree is `L` and that the value of `example7` for this tree is `false` leads to an inconsistency, because `example7 L` computes to `true` while it is assumed to be `false`. We can use the following tactic to exploit this kind of inconsistency:

```
intros H.
  H : true = false
  ============================
   1 = 3
discriminate H.
1 subgoal
...
  ============================
   example7 (N t1 t2) = false -> size (N t1 t2) = 3
```

The answer shows that the first goal was solved. The tactic `discriminate H` can be used whenever the hypothesis `H` is an assumption that asserts that two different constructors of an inductive type return equal values. Such an assumption is inconsistent and the tactic directly exploits this inconsistency to express that the case described in this goal can never happen.

Another important property of constructors of inductive types is that they are injective. The tactic to exploit this fact called `injection`. We illustrate it later in section 6.6.

For the second goal we still must do a case analysis on the values of `t1` and `t2`, we do not detail the proof but it can be completed with the following sequence of tactics.

```
destruct t1.
destruct t2.
3 subgoals

  ============================
   example7 (N L L) = false -> size (N L L) = 3

subgoal 2 is:
 example7 (N L (N t2_1 t2_2)) = false ->
 size (N L (N t2_1 t2_2)) = 3
subgoal 3 is:
 example7 (N (N t1_1 t1_2) t2) = false ->
  size (N (N t1_1 t1_2) t2) = 3
```

For the first goal, we can again ask that the functions in the goal should be computed.

```
simpl.
3 subgoals

  ============================
   false = false -> 3 = 3
```

Since the right hand side of the implication is a fact that is trivially true, we know that the automatic tactic of Coq can solve this goal (but using more elementary tactics from the table, we could use `intro` and `reflexivity` to prove it). The last two goals are solved in the same manner as the very first one, because `example7` cannot possibly have the value `false` for the arguments that are given in these goals.

```
auto.
intros H; discriminate H.
intros H; discriminate H.
Qed.
```

6.5 Proof by induction

The most general kind of proof that one can perform on inductive types is proof by induction. When we prove a property of the elements of an inductive type using a proof by induction, we actually consider a case for each constructor, as we did for proofs by cases. However there is a twist: when we consider a constructor that has arguments of the inductive type, we can assume that the property we want to establish holds for each of these arguments.

When we do goal directed proof, the induction principle is invoked by the `elim` tactic. To illustrate this tactic, we will prove a simple fact about the `flatten_aux` and `size` functions.

```
Lemma flatten_aux_size :
 forall t1 t2, size (flatten_aux t1 t2) = size t1 + size t2 + 1.
 induction t1.
  ============================
   forall t2 : bin, size (flatten_aux L t2) = size L + size t2 + 1

subgoal 2 is:
 forall t2 : bin,
 size (flatten_aux (N t1_1 t1_2) t2) =
 size (N t1_1 t1_2) + size t2 + 1
```

There are two subgoals, the first goal requires that we prove the property when the first argument of `flatten_aux` is `L`, the second one requires that we prove the property when the argument is `N t1_1 t1_2`. The proof progresses easily, using the definitions of the two functions, which are expanded when the Coq system executes the `simpl` tactic. We then obtain expressions that can be solved using the `ring` tactic.

```
intros t2.
simpl.
...
  ============================
   S (S (size t2)) = S (size t2 + 1)

ring.

  t1_1 : bin
  t1_2 : bin
  IHt1_1 : forall t2 : bin,
           size (flatten_aux t1_1 t2) = size t1_1 + size t2 + 1
  IHt1_2 : forall t2 : bin,
           size (flatten_aux t1_2 t2) = size t1_2 + size t2 + 1
  t2 : bin
  ============================
   size (flatten_aux (N t1_1 t1_2) t2) =
       size (N t1_1 t1_2) + size t2 + 1
```

In the second goal, we see that two induction hypotheses have been provided for the two subterms `t1_1` and `t1_2`. We can again, request that Coq computes symbolically the values of the two functions `size` and `flatten_aux`. Then, we can use the induction hypotheses.

```
intros t2; simpl.
...
  ============================
   size (flatten_aux t1_1 (flatten_aux t1_2 t2)) =
   S (size t1_1 + size t1_2 + size t2 + 1)

rewrite IHt1_1.
...
  ============================
   size t1_1 + size (flatten_aux t1_2 t2) + 1 =
   S (size t1_1 + size t1_2 + size t2 + 1)

rewrite IHt1_2.
...
  ============================
   size t1_1 + (size t1_2 + size t2 + 1) + 1 =
   S (size t1_1 + size t1_2 + size t2 + 1)

ring.
Proof completed.
Qed.
```

At this point, you should be able to perform your own proof by induction.

Exercise on `flatten` and `size` Prove

```
Lemma flatten_size : forall t, size (flatten t) = size t.
```

Hint: you should use `flatten_aux_size`, either with the tactic `apply` or with the tactic `rewrite`.

6.6 An example using `injection`

Let us now look at an example that illustrates the `injection` tactic. We want to prove that no tree can be a subterm of itself.

```
Lemma not_subterm_self_l : forall x y, ~ x = N x y.
  ============================
   forall x y : bin, x <> N x y
```

When reading this goal, we observe that negation of an equality is actually displayed as `_ <> _`. We will perform this proof by induction on `x`.

```
induction x.
2 subgoals

  ============================
   forall y : bin, L <> N L y

subgoal 2 is:
 forall y : bin, N x1 x2 <> N (N x1 x2) y
```

The first goal expresses that two constructors of the inductive type are different. This is a job for `discriminate`. The first goal is solved and we then see the second goal.

```
intros y; discriminate.
  IHx1 : forall y : bin, x1 <> N x1 y
  IHx2 : forall y : bin, x2 <> N x2 y
  ============================
   forall y : bin, N x1 x2 <> N (N x1 x2) y
```

Since the goal's conclusion is a negation, we can use the corresponding tactic from the table on page 13. This example also shows that the negation of a fact actually is represented by a function that says "this fact implies `False`".

```
intros y abs.
  abs : N x1 x2 = N (N x1 x2) y
  ============================
   False
```

The hypothesis `abs` expresses the equality between `N x1 x2` and `N (N x1 x2) y`. The constructor `N` is injective on both its arguments, so that this implies that `x1` is equal to `N x1 x2`. This reasoning step is expressed by the tactic `injection`:

```
injection abs.
  ============================
   x2 = y -> x1 = N x1 x2 -> False
```

Two new equations have been added to the goal's conclusion, but as premises of implications. These equations can be introduced as new hypotheses.

```
intros h2 h1.
  h2 : x2 = y
  h1 : x1 = N x1 x2
  ============================
   False
```

The hypothesis `h1` is actually contradictory with an instance of the hypothesis `IHx1`. We can express this with the following tactics.

```
assert (IHx1' : x1 <> N x1 x2).
 apply IHx1.
case IHx1'.
exact h1.
Qed.
```

7 Numbers in the Coq system

In the Coq system, most usual data-types are represented as inductive types and packages provide a variety of properties, functions, and theorems around these data-types. The package named `Arith` contains a host of theorems about natural numbers (numbers from 0 to infinity), which are described as an inductive type with `O` (representing 0) and `S` as constructors:

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

It also provides addition, multiplication, subtraction (with the special behavior that `x - y` is 0 when `x` is smaller than `y`). This package also provides a tactic `ring`, which solves equalities between expressions modulo associativity and commutativity of addition and multiplication and distributivity. For natural numbers, `ring` does not handle subtraction well, because of its special behavior. As we already mentioned, there is a syntactic facility, so that `3` actually represents `S (S (S O))`.

Since natural numbers are given as an inductive type, we can define recursive functions with natural numbers as input. The constraints on recursive calls makes that functions that call themselves recursively on the predecessor of the initial argument will be easy to define. For instance, the following definition works for the factorial function:

```
Fixpoint nat_fact (n:nat) : nat :=
  match n with O => 1 | S p => S p * nat_fact p end.
```

The following function works for the Fibonacci function:

```
Fixpoint fib (n:nat) : nat :=
  match n with
    O => 0
  | S q =>
    match q with
      O => 1
    | S p => fib p + fib q
    end
  end.
```

The package named `ZArith` provides two inductive data-types to represent integers. The first inductive type, named `positive`, follows a binary representation to model the positive integers (from 1 to infinity) and the type `Z` is described as a type with three constructors, one for positive numbers, one for negative numbers, and one for 0. The package also provides orders and basic operations: addition, subtraction, multiplication, division, square root. The tactic `ring` also works for integers (this time, subtraction is well supported) The tactic `omega` works equally well to solve problems consisting of collections of comparisons between linear formulas for both natural numbers of type `nat` and integers of type `Z`.

When working with integers instead of natural numbers it is handy to instruct Coq to interpret all numeric notations as notations concerning integers. This is done with the following command:

```
Open Scope Z_scope.
```

The type Z is not suited for structural recursion, so that it is not easy to define recursive functions with positive or negative integers as input. The main solution is to use well-founded recursion as described in [1], chapter 15. An alternative approach, less powerful but easy to understand, is to rely on an iterator that repeats the same operation a given number of times, where the number is described as an integer. This iterator is called `iter`.

```
Check iter.
iter : Z -> forall A : Type, (A -> A) -> A -> A
```

For instance, one can define the factorial function and test it as in the following dialog. We use an auxiliary function that takes a pair of numbers as input and returns a pair of numbers. When the input contains n and $n!$ the output contains $n+1$ and $(n+1)!$. We give this function as input to `iter`:

```
Definition fact_aux (n:Z) :=
 iter n (Z*Z) (fun p => (fst p + 1, snd p * (fst p + 1))) (0, 1).

Definition Z_fact (n:Z) := snd (fact_aux n).

Eval vm_compute in Z_fact 100.
93326...
```

Proving properties about functions defined using `iter` requires knowledge that goes beyond this short tutorial, but we give an example in the section that also contains the solutions to exercises at the end of this tutorial.

From now, we will revert to only manipulating natural numbers, we tell the Coq system to forget about the specification notations for integers.

```
Close Scope Z_scope.
```

8 Inductive properties

Inductive definitions can also be used to describe new predicates. A predicate on a type A is simply described as a function that takes as input elements of A and returns an element of the type of propositions, called `Prop`. The constructors of the inductive propositions are the theorems that characterize the predicate. In principle, every proof of a statement based on this predicate is obtained as a combination of the constructors, and only the constructors.

For example, a predicate on natural members that describes exactly the even numbers can be described by the following inductive definition.

```
Inductive even : nat -> Prop :=
  even0 : even 0
| evenS : forall x:nat, even x -> even (S (S x)).
```

Whenever x is a natural number, `even` x is a proposition. The theorem `even0` can be used to prove the proposition `even 0`. The theorem `evenS` can be used to deduce the proposition `even 2` from the proposition `even 0`. Repeated uses of `evenS` can be used to prove that other numbers satisfy the `even` predicate.

As for inductive types, the tactics `case`, `elim`, and `destruct` can be used to reason about assumptions concerning inductive propositions. These tactics again generate one goal for each of the constructors in the inductive definition. In addition, the tactics `elim` and `induction` generate induction hypotheses whenever a constructor uses the inductive proposition among its premises.

When a variable `x` satisfies an inductive property, it is often more efficient to prove properties about this variable using an induction on the inductive property than an induction on the variable itself. The following proof is an example:

```
Lemma even_mult : forall x, even x -> exists y, x = 2*y.
intros x H; elim H.
2 subgoals

  x : nat
  H : even x
  ============================
   exists y : nat, 0 = 2 * y
```

In the first goal, the system describes the case where `even x` is proved using the first constructor. In this case, the goal is displayed with the variable `x` replaced by `0`. we can prove this goal by providing 0 for the variable `y`.

```
exists 0; ring.
...
  ============================
   forall x0 : nat,
   even x0 -> (exists y : nat, x0 = 2 * y) ->
   exists y : nat, S (S x0) = 2 * y
```

Solving this goal reveals the next goal, which describes the case where `even x` is proved using the second constructor. In this case, there must be a variable `x0` such that `x` is `S (S x0)`, and the proposition even x0 must hold. In addition, the elim tactic generates an induction hypothesis corresponding to the proposition `even x0`. Thus `x0` is supposed to satisfy the property that we want to prove for `S (S x0)`. In this sense, the tactic `elim` creates an induction hypothesis, exactly like `induction` (you can actually use `induction` instead of `elim`, the two tactics are almost the same).

```
intros x0 Hevenx0 IHx.
...
  IHx : exists y : nat, x0 = 2 * y
  ============================
   exists y : nat, S (S x0) = 2 * y
```

Now, `IHx` is the induction hypothesis. It says that if `x` is the successor's successor of `x0` then we already know that there exists a value `y` that is the half of `x0`. We can use this value to provide the half of `S (S x0)`. Here are the tactics that complete the proof.

```
destruct IHx as [y Heq]; rewrite Heq.
exists (S y); ring.
Qed.
```

In this example, we used a variant of the `destruct` tactic that makes it possible to choose the name of the elements that `destruct` creates and introduces in the context.

The `inversion` tactic can be used to study the various ways in which the proof of a statement concerning an inductive proposition could have been made. This tactic analyses all the constructors of the inductive predicate, discards the ones that could not have been applied, and when some constructors could have been applied it creates a new goal where the premises of this constructor are added in the context. For instance, this tactic is perfectly suited to prove that 1 is not even, since no constructor can conclude to the proposition `even 1` (using `evenS` would require that `1 = S (S x)`, and using `even0` would require that `1 = 0`):

```
Lemma not_even_1 : ~even 1.
intros even1.
...
  even1 : even 1
  ============================
   False

inversion even1.
Qed.
```

Another example using the `inversion` tactic shows that when a number is even the predecessor of its predecessor is also even. In this statement, we write the first number as `S (S x)` and the predecessor of its predecessor as `x`.

```
Lemma even_inv : forall x, even (S (S x)) -> even x.
intros x H.
  H : even (S (S x))
  ============================
   even x
```

When the tactic analyzes the constructors of the `even` predicate, it recognizes that `even0` could not prove the statement of the hypothesis and only `evenS` could have been used. Therefore there must exist a variable `x0` such that `even x0` holds and `S (S x) = S (S x0)`; in other words, `even x` holds.

```
inversion H.
1 subgoal

  x : nat
  H : even (S (S x))
  x0 : nat
  H1 : even x
  H0 : x0 = x
  ============================
   even x
assumption.
Qed.
```

Please note that the statements of `even_inv` and `evenS` are similar but inverted. This is the reason for the tactic name.

Inductive properties can be used to express very complex notions. For instance, the semantics of a programming language can be defined as an inductive definition, using dozens of constructors, each one describing a an elementary step of computation.

9 Exercises

This section recapitulates the exercises from these notes. some are taken from [1].

Exercise on functions Write a function that takes five arguments and returns their sum, use `Check` to verify that your description is well-formed, use `Eval` to force its computation on a sample of values.

Exercise on lists, `map`, and `app` Define a function that takes as input a number n and returns a list with n elements, from 0 to $n-1$.

Exercise on sorting Define a function that takes a list as input and returns `true` when it has less than 2 elements or when the first element is smaller than the second one. Then define a function that is takes a list as input and returns `true` exactly when this list is sorted (Hint: when the list has at least two elements, the first element must be smaller than the second element and the tail must be sorted).

Exercise on counting Knowing that the Coq system provides a function `beq_nat` to compare two natural numbers, define a function that takes a natural number and a list and returns the number of times the natural number occurs in the list.

Exercise on logical connectives Prove the following theorems:

```
forall A B C:Prop, A/\(B/\C)->(A/\B)/\C
forall A B C D: Prop,(A->B)/\(C->D)/\A/\C -> B/\D
forall A: Prop, ~(A/\~A)
forall A B C: Prop, A\/(B\/C)->(A\/B)\/C
forall A B: Prop, (A\/B)/\~A -> B
```

Two benefits can be taken from this exercise. In a first step you should try using only the basic tactics given in the table page 13. In a second step, you can verify which of these statements are directly solved by the tactic `intuition`.

Exercise on universal quantification Prove

```
forall A:Type,forall P Q:A->Prop,
    (forall x, P x)\/(forall y, Q y)->forall x, P x\/Q x.
```

Exercise on addition, alternative definition We can define a new addition function on natural numbers:

```
Fixpoint add n m := match n with 0 => m | S p => add p (S m) end.
```

Prove the following statements (it is best to take them in order, as each can be used for the next one).

```
forall n m, add n (S m) = S (add n m)
forall n m, add (S n) m = S (add n m)
forall n m, add n m = n + m
```

Exercise on the sum of odd numbers The sum of the first n odd natural numbers is defined with the following function:

```
Fixpoint sum_odd_n (n:nat) : nat :=
  match n with  0 => 0 | S p => 1 + 2 * p + sum_odd_n p end.
```

Prove the following statement:

```
forall n:nat, sum_odd_n n = n*n
```

Exercise on datatype definition Define a datatype where there are three cases: a constant, a case where there are three fields, where the first field is a number and the next two fields are in the datatype being defined, and a case with four fields, where the first field is a boolean value and the three other fields are in the datatype being defined.

Exercise on `flatten` and `size` Prove

```
Lemma flatten_size : forall t, size (flatten t) = size t.
```

Hint: you should use `flatten_aux_size`, either with the tactic `apply` or with the tactic `rewrite`.

10 Solutions

The solutions to the numbered exercises are available from the Internet (on the site associated to the reference [1]).

10.1 functions

```
Check fun a b c d e => a + b + c + d + e.
Eval compute in (fun a b c d e => a + b + c + d + e) 1 2 3 4 5.
```

10.2 lists, map and app

```
Require Import List.

Fixpoint first_n_aux (n:nat)(m:nat) :=
 match n with 0 => nil | S p => m :: first_n_aux p (m+1) end.

Definition first_n (n:nat) := first_n_aux n 0.
```

10.3 sorting

```
Definition head_sorted (l : list nat) : bool :=
 match l with
   a::b::_ => leb a b
 | _ => true
 end.

Fixpoint sorted (l : list nat) : bool :=
match l with
  a::tl => if head_sorted (a::tl) then sorted tl else false
| nil => true
end.
```

10.4 counting

```
Fixpoint count (n : nat) (l : list nat) : nat :=
match l with
  nil => 0
| p::tl => if beq_nat n p then 1 + count n tl
end.
```

10.5 logical connectives

```
Lemma lc1 : forall A B C:Prop, A/\(B/\C) -> (A/\B)/\C.
intros A B C H; destruct H as [H H1]; destruct H1 as [H1 H2].
split. split. assumption. assumption. assumption.
Qed.

Lemma  lc2 : forall A B C D : Prop, (A->B)/\(C->D)/\A/\C->B/\D.
intros A B C D h'; destruct h' as [f [g [ha hc]]]; split.
 apply f; exact ha.  apply g; exact hc.
Qed.

Lemma lc3 : forall A : Prop, ~(A /\ ~A).
intros A. intros H. destruct H as [H1 H2]. destruct H2. exact H1.
Qed.

Lemma lc4 : forall A B C : Prop, A \/ (B \/ C) -> (A \/ B) \/ C.
intros A B C H. destruct H as [H | [H1 | H2]].
  left; left; exact H.
 left; right; exact H1.
right; exact H2.
Qed.

Lemma lc5 : forall A B : Prop, (A\/B)/\~A -> B.
intros A B h'; destruct h' as [[ha | hb] na].
 destruct na; exact ha.
exact hb.
Qed.
```

10.6 universal quantification

This exercise could also be solved directly using the tactic `firstorder`.

```
Lemma ex1 :
  forall A:Type, forall P Q:A->Prop,
  (forall x, P x) \/ (forall y, Q y) -> forall x, P x \/ Q x.
Proof.
 intros A P Q H.
 elim H.
 intros H1; left; apply H1.
 intros H2; right; apply H2.
Qed.
```

10.7 addition, alternative definition

```
Require Import Arith.

Fixpoint add n m := match n with 0 => m | S p => add p (S m) end.

Lemma add_n_S : forall n m, add n (S m) = S (add n m).
```

```
induction n; intros m; simpl.
 reflexivity.
rewrite IHn; reflexivity.
Qed.

Lemma add_S_n : forall n m, add (S n) m = S (add n m).
intros n m; simpl; rewrite add_n_S; reflexivity.
Qed.

Lemma add_plus : forall n m, add n m = n + m.
induction n; intros m.
 reflexivity.
rewrite add_S_n, IHn; reflexivity.
Qed.
```

10.8 sum of odd numbers

```
Require Import Arith.

Fixpoint sum_odd_n (n:nat) : nat :=
 match n with 0 => 0 | S p => 1 + 2 * p + sum_odd_n p end.

Lemma sum_odd_n_p : forall n, sum_odd_n n = n * n.
induction n.
 simpl. reflexivity.
simpl. rewrite IHn. ring.
Qed.
```

10.9 Datatype definition

```
Inductive tree3 : Type :=
  tree3_c
| tree3_3 (n : nat) (t1 t2 : tree3)
| tree3_4 (b : boolean) (t1 t2 t3 : tree3).
```

10.10 flatten

Here is the solution to the exercise on `flatten` and `size` (this re-uses the lemma `flatten_aux_size` proved in these notes).

```
Lemma flatten_size :  forall t, size(flatten t) = size t.
Proof.
intros t; elim t.
simpl. reflexivity.
intros t1 IH1 t2 IH2; simpl. rewrite flatten_aux_size. rewrite IH2. ring.
Qed.
```

10.11 Proof about `Z_fact`

We show that the factorial function computed on integers coincides with the factorial function computed on natural numbers. This is a simple proof by induction over natural numbers, but a lot of work is spent using the correspondances between integers and natural numbers. We need to first establish a lemma about the function `iter` to establish a correspondance with the corresponding iterator on natural numbers. This script uses tactics that have not been presented in this tutorial (`unfold`, `replace`,`trivial`) but whose documentation can be found in [2].

```
Lemma iter_nat_of_Z : forall n A f x, 0 <= n ->
  iter n A f x = iter_nat (Zabs_nat n) A f x.
intros n A f x; case n; auto.
  intros p _; unfold iter, Zabs_nat; apply iter_nat_of_P.
intros p abs; case abs; trivial.
Qed.

Lemma fact_aux_correct :
  forall n, fact_aux (Z_of_nat n) = (Z_of_nat n, Z_of_nat (nat_fact n)).
intros n; unfold fact_aux; rewrite iter_nat_of_Z; [ | apply Zle_0_nat].
induction n; [auto | ].
replace (nat_fact (S n)) with (S n * nat_fact n)%nat by auto.
assert(U : forall k A f x, iter_nat (S k) A f x=f (iter_nat k A f x)) by auto.
rewrite Zabs_nat_Z_of_nat in IHn |- *; rewrite U, IHn; unfold fst, snd.
rewrite inj_mult, Zmult_comm, inj_S; unfold Zsucc; auto.
Qed.

Lemma Z_fact_correct :
  forall n:nat, Z_fact (Z_of_nat n) = Z_of_nat (nat_fact n).
intros n; unfold Z_fact; rewrite fact_aux_correct; auto.
Qed.
```

References

[1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art:the Calculus of Inductive Constructions.* Springer-Verlag, 2004.

[2] The Coq development team. *The Coq proof Assistant Reference Manual,* Ecole Polytechnique, INRIA, Universit de Paris-Sud, 2004. `http://coq.inria.fr/doc/main.html`