Automated Reasoning 2020

Lecture 27: Interactive theorem proving

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2020-11-20

Limits of Automated solvers

There are several issues with SAT/SMT solvers.

- Not all logical questions can be automated.
- How can we trust SAT/SMT solvers, which are ensembles of optimizations? We need something more principled.

An answer to the above problems is interactive theorem proving.

Interactive theorem prover

- A software that helps you in proving theorems
  - Designed with solid mathematical foundation.
- It can automatically do a few simple operations
- For the rest user has to hint the actions.

Tactics

- Tactics are intuitive hints that suggest actions to the prover.
- The sequences of tactics are called proof scripts.

Content borrowed from tutorial: `https://coq.inria.fr/tutorial-nahas`

Topic 27.1

Coq basics

Introducing Coq proof script

```
Section Minimal_Logic.

Variables a b c : Prop.          (* declare proposition *)

Theorem trivial: a -> a.          (* simple theorem *)
Proof.                           (* start of proof *)
  intro proof_of_a.              (* introduces leading construct *)
  exact proof_of_a.
Qed.
```

This is not a proof!
This is a proof script.

Repeated intro and reuse results

```
Theorem reuse: b -> a -> a.      (* slightly complex theorem *)
Proof.
  intro proof_of_b.
  exact trivial.
Qed.
```

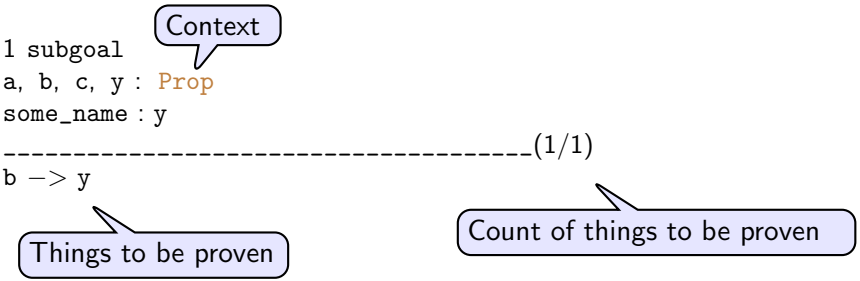
```
Theorem no_reuse: b -> a -> a.      (* using intros *)
Proof.
  intros.
  exact H.
Qed.
```

Introducing forall

```
(* forall and implications are treated in a similar way!! *)
Theorem demoAll: (forall x: Prop, x -> b -> x).
Proof.
  intros y. (* first introduce quantified variable *)
  intros some_name pb.
  exact some_name.
Qed.
```

Commentary: Declaring variables outside of the theorem and declaring forall seems to have same effect. Only difference is the scope.

Proof state


Context
1 subgoal
a, b, c, y : Prop
some_name : y
(1/1)
b -> y
Things to be proven
Count of things to be proven

Forward proving

```
(* learning pose *)
Theorem non_trivial: a -> (a->b) -> b.
Proof.
  intros pa pab.
  pose (pb := pab pa).      (* pose applies known facts! *)
  exact pb.
Qed.
```

Exercise 27.1

Prove the following theorem.

```
Theorem ex_1 : (forall A B C : Prop, A -> (A->B) -> (A->B->C) -> C).
```

We can see the proof in progress!!

```
Theorem non_trivial: a -> (a->b) -> b.
Proof.
  Show Proof.
  intros pa pab.
  Show Proof.          (* shows the current proof term *)
  pose (pb := pab pa).
  Show Proof.
  exact pb.
Qed.
```

Proof state and proof term (contd.)

In the previous example, the following is the proof state at the second Show Proof.

```
1 subgoal
a, b, c : Prop
pa : a
pab : a -> b
------------------------------------(1/1)
b
```

Proof term at the same time.

```
(fun (pa : a) (pab : a -> b) => ?Goal)
```

Proof state and proof term

Thereafter, the following is the proof state after applying `pose (pb := pab pa).`

```
1 subgoal
a, b, c : Prop
pa : a
pab : a -> b
pb := pab pa : b
------------------------------------(1/1)
b
```

Proof term at the same time.

```
(fun (pa : a) (pab : a -> b) =>
   let p_b := pab pa : b in
   ?Goal
)
```

Reverse proving

```
(* learning refine *)
Theorem non_trivial: a -> (a->b) -> b.
Proof.
  intros pa pab.
  refine (pab _).    (* what do we need to prove the goal? *)
  exact pa.
Qed.
```

Exercise 27.2

Prove the following theorem using refine.

```
Theorem ex_1 : (forall A B C : Prop, A -> (A->B) -> (A->B->C) -> C).
```

Reverse proving using apply

```
(* learning apply *)
Theorem non_trivial: a -> (a->b) -> b.
Proof.
  intros pa pab.
  apply pab.       (* same as refine without explicit patterns *)
  exact pa.
Qed.
```

Coq basics

- Proofs are functional programs
  - Curry-Howard isomorphism
- Universal quantifiers and implications are base connectives
  - Almost everything can be written using them except "negation"
- Let us see what we need to add to encode negation.

Topic 27.2

Negation and False

Negation is defined using "False"

- We can define $\neg A$ as $A \Rightarrow \mathit{False}$
- We need to define `False` and also `True`
- They are defined using the following constructs:

```
Inductive False : Prop := .

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

Inductive is used to define instances of types.

`False` has no way of proving.

`True` can be proven using `I`, which can be always proven.

Commentary: Inductive is yet another fundamental idea in Coq. Now take it as presented. Let us be more familiar with the system before having clearer understanding.

Negation is defined using "False" II

- Negation is defined as follows

```
Definition not (A:Prop) := A -> False.  (* macro that defines not *)

Notation "~ x" := (not x) : type_scope. (* syntactic sugar for not *)
```

Understanding "False" and "True"

Our usual Boolean will be defined later.

In Coq, they actually mean unprovable and provable.

```
Theorem True_can_be_proven : True.
Proof.
  exact I. (* True can be matched with I, which is always there*)
Qed.
```

```
Theorem False_can_never_be_proven : ~ False.
Proof.
  unfold not. (* unfold expands predefined macros *)
  intros pf.
  exact pf.
Qed.
```

Let us prove a few facts about True and False

We have setup a world of True and False. Do they match our intuition?

```
Theorem thm_T_imp_T : True -> True.
Proof.
  intros proof_of_True.
  exact I.
Qed.

Theorem thm_F_imp_F : False -> False.
Proof.
  intros pf.
  case pf. (* "exact pf." works, but is not recommended. *)
Qed.
```

Exercise 27.3

Prove the following theorems.

```
Theorem thm_F_imp_T : False -> True.
Theorem thm_T_imp_F : ~ (True -> False).
```

Reducto ad absurdium

If you can derive contradiction, you can derive anything.

```
Theorem absurd : forall A C : Prop, A -> ~ A -> C.
Proof.
  intros A C.
  intros pa p_nota.
  unfold not in p_nota.
  pose (pf := p_nota pa).
  case pf.
Qed.
```

Topic 27.3

Defining Or/And over Prop

or/and are defined as follows

Back to propositions....

or can be constructed in two ways

```
Inductive or ( A B : Prop ) : Prop :=
| or_introl : A -> A \/ B
| or_intror : B -> A \/ B
where "A \/ B" := (or A B) : type_scope.
```

and has only one construction

```
Inductive and ( A B : Prop ) : Prop :=
  conj : A -> B -> A /\ B
where "A /\ B" := (and A B) : type_scope.
```

/\ and \/ are shorthands.

A few theorems on or

```
Theorem left_or : (forall A B : Prop, A -> A \/ B).
Proof.
intros A B.
intros pa.
pose (pab := or_introl pa : A \/ B).
exact pab.
Qed.
```

pose needs type declaration because `or_introl` does not have full type information.

Exercise 27.4

Prove the following theorems using refine or apply.

```
Theorem right_or : (forall A B : Prop, B -> A \/ B).
Theorem both_and : (forall A B : Prop, A -> B -> A /\ B).
```

Or/And commutes

```
Theorem or_commutes : (forall A B, A \/ B -> B \/ A).
Proof.
  intros A B.
  intros A_or_B.
  case A_or_B. (* creates two subgoals for each constructor of \/ *)
    intros pa.          (*suppose A_or_B is (or_introl proof_of_A) *)
    refine (or_intror _).
      exact pa.

    intros pb.          (*suppose A_or_B is (or_intror proof_of_B) *)
    refine (or_introl _).
      exact pb.
Qed.
```

Exercise 27.5

Prove the following theorem.

```
Theorem and_commutes : (forall A B, A /\ B -> B /\ A).
```

destruct tactic

```
Theorem and_commutes__again : (forall A B, A /\ B -> B /\ A).
Proof.
intros A B.
intros A_and_B.
destruct A_and_B as [ pa pb].
refine (conj _ _).
exact pa.

exact pb.
Qed.
```

destruct combines the work of case and subsequent intros.

Topic 27.4

Booleans

The Booleans we know!

Boolean is already defined as follows.

```
Inductive bool : Set :=
| true : bool
| false : bool.
```

To import the above definitions and many more, we need to add.

```
Require Import Bool.
```

Commentary: Set is yet another fundamental concept in the Coq system.

The Booleans we know!(II)

The import loads the following two import functions

```
(* Defines equality over Booleans *)
Definition eqb (b1 b2:bool) : bool :=
match b1, b2 with
| true, true => true
| true, false => false
| false, true => false
| false, false => true
end.
```

```
(* maps Booleans to provability *)
Definition Is_true (b:bool) :=
match b with
| true => True
| false => False
end.
```

Let us prove some theorems on Booleans

```
Theorem true_is_True: Is_true true.
Proof.
  simpl.  (* executes the head function at the goal *)
  exact I.
Qed.

Theorem not_eqb_true_false: ~ (Is_true (eqb true false)).
Proof.
  simpl.
  exact False_cannot_be_proven.
Qed.
```

Another simple theorem on Booleans

```
Theorem eqb_a_a : (forall a : bool, Is_true (eqb a a)).
Proof.
  intros a.
  case a.    (* creates two subgoals for each value of bool *)
    simpl.
    exact I.

    simpl.
    exact I.
Qed.
```

case only works if a is not referred anywhere else in the context.

Exercise 27.6

Prove the following theorem.

```
Theorem ex_4: (forall a:bool, (Is_true (eqb a true)) -> (Is_true a)).
```

Topic 27.5

Exists

Existence

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

Notation "'exists' x .. y , p" := (ex (fun x => .. (ex (fun y => p)) ..))
  (at level 200, x binder, right associativity,
   format "'[' 'exists' '/ ' x .. y , '/ ' p ']'")
   : type_scope.
```

- For any x:A if we can show P x, then we have ex P
- ex constructor is ex_intro
- exists is shorthanded notation for ex

A proof for existence!

How coq knows types of a and b?

```
Theorem thm_forall_exists : (forall b, (exists a, Is_true(eqb a b) ) ).
Proof.
  intros b.
  case b.
(* b is true *)
    pose (witness := true).
    refine (ex_intro _ witness _).
      simpl.
      exact I.

(* b is false *)
    pose (witness := false).
    refine (ex_intro _ witness _).
      simpl.
      exact I.
Qed.
```

An important property of exists

```
Theorem fe: (forall P : Set->Prop,(forall x, ~ (P x)) -> ~ (exists x, P x)).
Proof.
  intros P.
  intros forall_x_not_Px.
  unfold not.
  intros exists_x_Px.
  destruct exists_x_Px as [ witness proof_of_Pwitness].
  pose (not_Pwitness := forall_x_not_Px witness).
  unfold not in not_Pwitness.
  pose (proof_of_False := not_Pwitness proof_of_Pwitness).
  case proof_of_False.
Qed.
```

Exercise 27.7

Prove the following theorem.

```
Theorem ef: (forall P : Set->Prop,~ (exists x, P x) -> (forall x, ~ (P x))).
```

Topic 27.6

Equality

Equality and inequality

- Equality is also defined object.

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

where "x = y :> A" := (@eq A x y) : type_scope.

Notation "x = y" := (x = y :>_) : type_scope.
```

- Once two things become equal they are same constants.

- Inequality is defined as follows.

```
Notation "x <> y :> T" := (~ x = y :>T) : type_scope.
Notation "x <> y" := (x <> y :>_) : type_scope.
```

Proving with equality

```
Theorem thm_eq_sym : (forall x y : Set, x = y -> y = x).
Proof.
  intros x y x_y.
  destruct x_y as [].
  exact (eq_refl x).
Qed.
```

Topic 27.7

Induction

Defining natural numbers

Let us see our first true inductive definition that defines natural numbers.

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

- `O` is zero and `S` is successor function.
- Natural numbers as terms over the symbols, e.g., 3 is `S(S(S(O)))`.
- Addition is defined as following function.

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

where "n + m" := (plus n m) : nat_scope.
```

Numbers and terms

Numbers are defined as terms. However, one can write them as usual numbers and they are interpreted as the terms.

```
Theorem plus_2_3 : (S (S 0)) + (S (S (S 0))) = (S (S (S (S (S 0))))).
Proof.
 simpl.
 exact (eq_refl 5).
Qed.
```

Proving with inductive definitions.

```
Theorem plus_0_n : (forall n, 0 + n = n).
Proof.
  intros n.
  simpl.
  exact (eq_refl n).
Qed.
```

Can we prove the following using the same tactics?

```
Theorem plus_n_0 : (forall n, n + 0 = n).
```

Induction principle

In Coq induction principle for natural numbers is defined as follows.

```
nat_ind
 : forall P : nat -> Type,
     P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Applying induction principle

```
Theorem plus_n_O : (forall n, n + 0 = n).
Proof.
  intros n.
  elim n. (* applies induction hypothesis on the nat definition*)
    (* first subgoal: base case *)
    simpl.
    exact (eq_refl 0).

    (* second subgoal: inductive case *)
    intros n'. (* must use fresh name to instantiate *)
    intros inductive_hypothesis.
    simpl.
    rewrite inductive_hypothesis.
    exact (eq_refl (S n')).
Qed.
```

Exercise 27.8

Prove the following theorem.

```
Theorem plus_sym: (forall n m, n + m = m + n).
```

Topic 27.8

Datatypes

Defining datatypes

Like natural numbers, other objects are also defined inductively. For example,

```
Inductive list (A : Type) : Type :=
| nil : list A
| cons : A -> list A -> list A.

Infix "::" := cons (at level 60, right associativity) : list_scope.
```

- `nil` is for empty list and `cons` extends a list.
- `::` is shorthand for `cons`.
- List length is defined as the following function.

```
Definition length (A : Type) : list A -> nat :=
fix length l :=
match l with
| nil => 0
| _ :: l' => S (length l')
end.
```

Other functions on lists

```
Definition hd (A : Type) (default : A) (l : list A) :=
  match l with
    | nil => default
    | x :: _ => x
  end.

Definition tl (A : Type) (l:list A) :=
  match l with
    | nil => nil
    | a :: m => m
    end.

Definition app (A : Type) : list A -> list A -> list A :=
  fix app l m :=
  match l with
   | nil => m
   | a :: l1 => a :: app l1 m
  end.
Infix "++" := app (right associativity, at level 60) : list_scope.
```

A simple theorem on lists

```
Theorem hd_tl :
   (forall A:Type,
   (forall (default : A) (x : A) (lst : list A),
   (hd A default (x::lst)) ::  (tl A (x:: lst)) = (x ::  lst))).
Proof.
  intros A.
  intros default x lst.
  simpl.
  exact (eq_refl (x::lst)).
Qed.
```

Topic 27.9

Problem

Propositions

Exercise 27.9

Prove the following theorems.

```
Theorem p1 : (forall A B : Prop, (A -> B ) -> (~ B -> ~ A) ).
```

```
Theorem p2 : (forall A B : Prop, (A \/ B )/\ (C\/D) -> ( (A/\C) \/ B \/ D ) ).
```

Odd and even

Exercise 27.10

Using Coq formalize definition of odd and even numbers. Subsequently, prove the following theorems

- 5 is an odd number.
- No number is both odd and even.
- Between every two odd numbers there is at least one even number.
- Between every two consecutive even numbers there is exactly one odd number.
- There is no largest even number.

Proving arithmetic correct!

Exercise 27.11

Using Coq formalize following definitions

- Define natural numbers as list of digits, e.g., `zero::nine::one` is 190.
- Define primary school algorithms for addition, subtraction, and multiplication over the lists.

And prove that the algorithms are correct with respect to the native natural numbers and arithmetic operations defined in Coq.

For clarifications: please contact the instructor.

End of Lecture 27