Proof assistants as a tool for thought

Katherine Ye
@hypotext
circa 1700
Disputants unable to agree would not waste much time in futile argument...

*Leibniz (Harrison)*
Calculemus!
1. universal language
2. calculus for reasoning
Proof assistant: Coq
1. universal language
2. calculus for reasoning
1. universal language
2. calculus for reasoning
3. rich environment
High-assurance cryptography
New foundations for math
Program synthesis
Verifying hardware
Proofs as stories
Formally verifying that God exists
...

Verified correctness and security of OpenSSL HMAC

To appear in 24th Usenix Security Symposium, August 12, 2015

Lennart Beringer  
Princeton Univ.

Adam Petcher  
Harvard Univ. and  
MIT Lincoln Laboratory

Katherine Q. Ye  
Princeton Univ.

Andrew W. Appel  
Princeton Univ.
Example 1: math, induction

Credit to “Software Foundations,” Pierce et al.
Say we want to prove something about adding natural numbers.
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.

Natural number = either 0
or 1 + a nat
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.

Fixpoint plus (x y : nat) :=
  match x with
  | O => y
  | S x' => S (plus x' y)
  end.

\[
\begin{align*}
0 + y &= y \\
(l + x') + y &= l + (x' + y)
\end{align*}
\]
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.

Fixpoint plus (x y : nat) :=
  match x with
  | O => y
  | O => y
  | S x' => S (plus x' y)
  end.

"Unit tests"

(* O = 0, S O = 1, S (S O) = 2 *)
Eval compute in (plus O O). (* 0 + 0 = 0 *)
Inductive nat : Set :=
  | O : nat
  | S : nat -> nat.

Fixpoint plus (x y : nat) :=
  match x with
  | O => y
  | S x' => S (plus x' y)
  end.

(* 0 = 0, S 0 = 1, S (S 0) = 2 *)
Eval compute in (plus 0 0). (* 0 + 0 = 0 *)
Eval compute in (plus (S 0) 0). (* 1 + 0 = 1 *)
Eval compute in (plus 0 (S 0)). (* 0 + 1 = 1 *)
Eval compute in (plus (S 0) (S 0)). (* 1 + 1 = 2 *)
“Unit tests” aren't enough. Now we want prove that our computational `plus` satisfies the properties of the mathematical +.
0 is a left identity

Theorem add0_left_id :
forall (n : nat), plus O n = n.
Proof.
Theorem add0_left_id :
  \forall (n : \text{nat}), \text{plus} \ O \ n = n.
Proof.

\text{Fixpoint plus} \ (x \ y : \text{nat}) :=
\begin{align*}
\text{match} \ x \ \text{with} \\
& | \ O \Rightarrow y \\
& | \ S \ x' \Rightarrow S \ (\text{plus} \ x' \ y)
\end{align*}
\text{end.}
Theorem add0_left_id :
  forall (n : nat), plus O n = n.
Proof.
  intros n.
  simpl.
  reflexivity.
Qed.

Fixpoint plus (x y : nat) :=
  match x with
  | 0 => y
  | S x' => S (plus x' y)
  end.
0 is a right identity

Theorem add0_right_id :
  forall (n : nat), plus n 0 = n.
Proof.

Fixpoint plus (x y : nat) :=
  match x with
  | 0 => y
  | S x' => S (plus x' y)
end.
Theorem add0_right_id :
  forall (n : nat), plus n 0 = n.

Proof.
  intros n.
Theorem add0_right_id :
  forall (n : nat), plus n 0 = n.
Proof.
  intros n.
  induction n as [ | n'].
Theorem add0_right_id :
  forall (n : nat), plus n 0 = n.
Proof.
  intros n.
  induction n as [ | n'].

  (* Base case: n = 0 *)

  plus 0 0 = 0
Theorem add0_right_id :
  forall (n : nat), plus n 0 = n.
Proof.
  intros n.
  induction n as [ | n'].

  (* Base case: n = 0 *)
  - reflexivity.
Theorem add0_right_id :
  \( \forall (n : \text{nat}), \text{plus } n \ 0 = n \).
Proof.
  intros n.
  induction n as [ | n'].

  (* Base case: n = 0 *)
  - reflexivity.

  (* Inductive case: *)
  Given: \( n' + 0 = n' \)
  Show: \( (1 + n') + 0 = (1 + n') \)
Theorem add0_right_id:
  forall (n : nat), plus n 0 = n.

Proof.
  intros n.
  induction n as [ | n'].
  (* Base case: n = 0 *)
  - reflexivity.
  (* Inductive case: *)
  Given: n' + 0 = n'
  Show: (1 + n') + 0 = (1 + n')
  - simpl.
  (* Goal: (1 + n') + 0 = 1 + n' becomes 1 + (n' + 0) = 1 + n' *)
Theorem \texttt{add0_right_id}:
\[
\text{forall } (n : \text{nat}), \text{ plus } n \ 0 = n.
\]

Proof.
\begin{itemize}
\item \texttt{intros n}.
\item \texttt{induction n as [ | n'] }.
\end{itemize}

(* Base case: \( n = 0 \) *)
- \texttt{reflexivity}.

(* Inductive case: 
  Given: \( n' + 0 = n' \)
  Show: \( (1 + n') + 0 = (1 + n') \) *)
- \texttt{simpl}.
  (* Goal: \( (1 + n') + 0 = 1 + n' \) becomes \( 1 + (n' + 0) = 1 + n' \) *)
  \texttt{rewrite \rightarrow IHn'}. 

1 focused subgoals (unfocused: 0)
, subgoal 1 (ID 49)

\( n' : \text{nat} \)
\( \text{IHn'} : \text{plus } n' \ 0 = n' \)

\( S \ n' = S \ n' \)
Theorem add0_right_id:
  \forall (n : \text{nat}), \text{plus } n \ 0 = n.
Proof.
  intros n.
  induction n as [ | n'].

  (* Base case: n = 0 *)
  - reflexivity.

  (* Inductive case: *)
  Given: n' + 0 = n'
  Show: (1 + n') + 0 = (1 + n') *
  - simpl.
  (* Goal: (1 + n') + 0 = 1 + n' becomes 1 + (n' + 0) = 1 + n' *)
  rewrite -> IHn'.
  reflexivity.
Qed.
Theorem \textit{add0\_right\_id}:
\[\forall n : \text{nat}, \; \text{plus} n 0 = n.\]

Proof.
intros n.
induction n as [ _ n'].

(* Base case: \(n = 0\) *)
- reflexivity.

(* Inductive case: 
Given: \(n' + 0 = n'\)
Show: \((1 + n') + 0 = (1 + n')\) *)
- simpl.
  (* Goal: \((1 + n') + 0 = 1 + n'\)
becomes \(1 + (n' + 0) = 1 + n'\) *)
rewrite -> IHn'.
reflexivity.

Qed.
Exercise for the reader:

Theorem plus_commutativity :
forall (n m : nat), plus n m = plus m n.
Example 2: large real-world verification
Verified correctness and security of OpenSSL HMAC

To appear in 24th Usenix Security Symposium, August 12, 2015

Lennart Beringer  
Princeton Univ.

Adam Petcher  
Harvard Univ. and  
MIT Lincoln Laboratory

Katherine Q. Ye  
Princeton Univ.

Andrew W. Appel  
Princeton Univ.
1. Functional correctness
   (logic, program analysis)

2. Security
   (math, probability, program analysis)
Figure 1: Architecture of our assurance case.
5+ months of work by 4 authors

~15,000 lines of Coq code, most of which will not be read by other people
Time for cognitive dissonance!
The house believes that Coq is an incredible tool for thought.
The house believes that Coq is a terrible tool for thought.
The house believes that Coq is an incredible tool for thought.
Built on top of:

written notation
Built on top of:

written notation
text editors
Built on top of:

written notation
text editors
Gallina
Built on top of:

written notation
text editors
Gallina
Ltac
Built on top of:

written notation
text editors
Gallina
Ltac
checker
Built on top of:

written notation
text editors
Gallina
Ltac
checker
REPL/IDE
Started from the bottom, now we here
Proof entrepreneur:
fail fast,
minimum viable proof...
Proof state bookkeeping: assumptions, definitions, goals

1 focused subgoals (unfocused: 0)
, subgoal 1 (ID 48)

\n' : nat
IHn' : plus n' 0 = n'

S (plus n' 0) = S n'
Computation, evaluation, automation

(* 0 = 0, S 0 = 1, S (S 0) = 2 *)
Eval compute in (plus 0 0). (* 0 + 0 = 0 *)
Eval compute in (plus (S 0) 0). (* 1 + 0 = 1 *)
Eval compute in (plus 0 (S 0)). (* 0 + 1 = 1 *)
Eval compute in (plus (S 0) (S 0)). (* 1 + 1 = 2 *)

U:%-%  *goals*
1 = S (S 0)
2 : nat
Try

Fail → Fix

Maybe right?

Brain says:
this isn’t right... I think

Maybe right?
Coq says:
This is locally wrong!
Wrong type /
Can’t prove it /
Can’t use tactic /
Strange computation
Sketching with lemmas
Goal
Lemma 1
Lemma 2
Lemma 3
Lemma 4
Lemma 5

Coq: OK!
admitted

Lemma 1
Lemma 2
Lemma 3
Lemma 4
Lemma 5

Coq: OK!
admitted

Coq: OK!
admitted

Coq: OK!
admitted

Coq: OK!
admitted
Goal

Lemma 1

Lemma 2

Lemma 3

Lemma 4

Lemma 5

Coq: OK!

QED

Coq: OK!

QED

Coq: OK!

QED

Coq: OK!

QED

Coq: OK!

QED
Math as a game
You don’t have to remember all the rules yourself.
\[ x + 20 = y \]

\[ x = y - 20 \]
$x + 20 = y$

$x + 20 - 20 = y - 20$

$x = y - 20$
Civilization advances by extending the number of important operations which can be performed without thinking about them.

Whitehead
Goal: beat the level.
Proof state: inventory.
Tactics: moves.
Checker: walls.
We need Coq in order to keep doing math.
A technical argument by a trusted author ... is **hardly ever checked in detail.**

Voevodsky

*Fields medalist*
The only real long-term solution to the problem is to **start using computers in the verification of mathematical reasoning.**

Voevodsky

*Fields medalist*
We need proofs that are less error-prone and more ... \textit{mechanically verifiable}.

\textit{Bellare cryptographer}
Many proofs in cryptography have become essentially unverifiable. Our field may be approaching a crisis of rigor.

Bellare cryptographer

(100+ citations!)
Pro-Coq:

1. Programmer affordances
2. Math as a game
3. “Crisis of rigor”
The house believes that
Coq is a terrible tool
for thought.
Calculemus?
Games are designed for humans to solve.

Math isn’t!
What could go wrong?
Your theorem is:

too specific, so you need to generalize
Your theorem is:

too specific, so you need to generalize
straight-up wrong, so you need a counterexample
Your theorem is:

too specific, so you need to generalize
straight-up wrong, so you need a counterexample
true, but you need to be creative
Your theorem is:

too specific, so you need to generalize
straight-up wrong, so you need a counterexample
true, but you need to be creative
true, but you discarded the One Ring
Work on paper first, then in computer :(
Coq blindfolds the intuition.
We’re not “Coq natives”!
Built on top of:

- written notation
- text editors
- Gallina
- Ltac
- checker
- IDE
Started from the bottom...
now we’re back.
REPLs are alive and make us reactive
Try

Brain says:
this isn’t right...
I think.

Fail

Fix
Paper is calm and makes us active
Paper forces us to figure out what’s going on, formulate a hypothesis
So, Coq makes proofs harder to write.
...and harder to read!
“write-only”
Theorem PRF_A_randomFunc_eq_until_bad :
  comp_spec
  (fun y1 y2 : bool * (list (D * Bvector eta)) =>
   hasDups_ (fst (split (snd y1))) =
   hasDups_ (fst (split (snd y2))) /
   (hasDups_ (fst (split (snd y1))) = false =>
    snd y1 = snd y2 /
    fst y1 = fst y2))
  (PRF_A__ randomFunc_withDups nil)
  (PRF_A__
   (fun (ls : list (D * Bvector eta)) (x : D) =>
    r <- \{ \emptyset , 1 \}^\text{eta}; ret (r, (x, r) :: ls) nil).

Proof.
  eapply (fcf_oracle_eq_until_bad
    (fun x => hasDups_ (fst (split x)))
    (fun x => hasDups_ (fst (split x))) eq);

  intuition.
  - apply PRF_A_wf.
  - unfold randomFunc_withDups.
  destruct (arrayLookup D_EqDec a b);
  fcf_well_formed.
  - fcf_well_formed.
  - subst.
  unfold randomFunc_withDups.
  case_eq (arrayLookup _ x2 a); intuition.

* fcf_irr_r.
  fcf_simp.
  fcf_spec_ret; simpl.

  remember (split x2) as z.
  destruct z.
  simpl in *.
  trivial.
  simpl in *.
  remember (split x2) as z.
  destruct z.
  simpl in *.
  destruct (in_dec (EqDec_dec D_EqDec a \emptyset))%; intuition.
  discriminate.
  rewrite notInArrayLookupNone in H.
  discriminate.
  intuition.
  rewrite unzip_eq_split in H3.
  remember (split x2) as z.
  destruct z.
  pairInv.
  simpl in *.
  intuition.

  simpl in *.
what are we trying to prove??

Theorem PRF_A_randomFunc_eq_until_bad :
  comp_spec
  (fun y1 y2 : bool * (list (D * Bvector eta)) =>
    hasDups _ (fst (split (snd y1))) =
    hasDups _ (fst (split (snd y2))) ∧
    (hasDups _ (fst (split (snd y1))) = false ->
    snd y1 = snd y2 ∧ fst y1 = fst y2))
  (PRF_A _ _ randomFunc_withDups nil)
  (PRF_A _ _
    (fun (ls : list (D * Bvector eta)) (x : D) =>
    r <$> { 0, 1 }^eta; ret (r, (x, r) :: ls)) nil).

Proof.
  eapply (fcf_oracle_eq_until_bad
    (fun x => hasDups _ (fst (split x)))
    (fun x => hasDups _ (fst (split x))) eq);

  intuition.
  - apply PRF_A_wf.
  - unfold randomFunc_withDups.
  destruct (arrayLookup D_EqDec a b);
  fcf_well_formed.
  - fcf_well_formed.
  - subst.
  unfold randomFunc_withDups.
  case_eq (arrayLookup _ x2 a); intuition.

* fcf_irr_r.
  fcf_simp.
  fcf_spec_ret; simpl.

  remember (split x2) as z.
  destruct z.
  simpl in *.
  trivial.
  simpl in *.
  remember (split x2) as z.
  destruct z.
  simpl in *.
  destruct (in_dec (EqDec_dec D_EqDec a 10)); intuition.
  discriminate.
  rewrite notInArrayLookupNone in H.
  discriminate.
  intuition.
  rewrite unzip_eq_split in H3.
  remember (split x2) as z.
  destruct z.
  pairInv.
  simpl in *.
  intuition.
  simpl in *.
Theorem PRF_A_randomFunc_eq_until_bad :
  comp_spec
  (fun y1 y2 : bool * (list (D * Bvector eta)) =>
    hasDups _ (fst (split (snd y1))) =
    hasDups _ (fst (split (snd y2))) /
    (hasDups _ (fst (split (snd y1))) = false ->
    snd y1 = snd y2 /
    fst y1 = fst y2))
  (PRF_A _ _ randomFunc_withDups nil)
  (PRF_A _ _
    (fun (ls : list (D * Bvector eta)) (x : D) =>
    r <- $ { 0, 1 }^eta; ret (r, (x, r) :: ls)) nil).

Proof.
eapply (fcf_oracle_eq_until_bad
  (fun x => hasDups _ (fst (split x)))
  (fun x => hasDups _ (fst (split x)) eq);

intuition.
- apply PRF_A_wf.
- unfold randomFunc_withDups.
destruct (arrayLookup D_EqDec a b);
  fcf_well_formed.
- fcf_well_formed.
- subst.
  unfold randomFunc_withDups.
case_eq (arrayLookup _ x2 a); intuition.

* fcf_irr_r.
  fcf_simp.
  fcf_spec_ret; simpl.

  remember (split x2) as z.
destruct z.
simpl in *.
  trivial.
simpl in *.
  remember (split x2) as z.
destruct z.
simpl in *.
destruct (in_dec (EqDec_dec D_EqDec a l0)); intuition.
discriminate.
rewrite notInArrayLookupNone in H.
discriminate.
intuition.
rewrite unzip_eq_split in H3.
remember (split x2) as z.
destruct z.
pairInv.
simpl in *.
  intuition.
simpl in *.

Check fcf_oracle_eq_until_bad.
Locate fcf_oracle_eq_until_bad.
Why applied with these arguments?
Theorem PRF_A_randomFunc_eq_until_bad :
  comp_spec
  (fun y1 y2 : bool * (list (D * Bvector eta)) =>
    hasDups _ (fst (split (snd y1))) =
    hasDups _ (fst (split (snd y2))) /
    (hasDups _ (fst (split (snd y1))) = false ->
     snd y1 = snd y2 /
     fst y1 = fst y2))
  (PRF_A _ randomFunc_withDups nil)
  (PRF_A _
    (fun (ls : list (D * Bvector eta)) (x : D) =>
       r <- $ { 0 , 1 }^eta; ret (r, (x, r) :: ls)) nil).

Proof.
eapply (fcf_oracle_eq_until_bad
  (fun x => hasDups _ (fst (split x)))
  (fun x => hasDups _ (fst (split x))) eq);

intuition.
- apply PRF_A_wf.
- unfold randomFunc_withDups.
destruct (arrayLookup D_EqDec a b);
fcf_well_formed.
- fcf_well_formed.
- subst.
unfold randomFunc_withDups.
case_eq (arrayLookup _ x2 a); intuition.

* fcf_irr_r.
  fcf_simp.
  fcf_spec_ret; simpl.

  remember (split x2) as z.
destruct z.
simpl in *.
trivial.
simpl in *.
remember (split x2) as z.
destruct z.
simpl in *.
destruct (in_dec (EqDec_dec D_EqDec a l0)); intuition.
discriminate.
rewrite notInArrayLookupNone in H.
discriminate.
intuition.
rewrite unzip_eq_split in H3.
remember (split x2) as z.
destruct z.
pairInv.
simpl in *.
intuition.
simpl in *.
What hypothesis did I use?
Theorem PRF_A_randomFunc_eq_until_bad:

```
comp_spec
  (fun y1 y2 : bool * (list (D * Bvector eta)) =>
    hasDups_ (fst (split (snd y1))) =
    hasDups_ (fst (split (snd y2))) ∧
    (hasDups_ (fst (split (snd y1))) = false ⇒
    snd y1 = snd y2 ∧ fst y1 = fst y2))
(PRFl_A _ _ randomFunc_withDups nil)
(PRFl_A _ _
  (fun (ls : list (D * Bvector eta)) (x : D) =>
    r <$> { 0, 1 }^eta; ret (r, (x, r) :: ls)) nil).
```

Proof.
```
eapply (fcf_oracle_eq_until_bad
  (fun x ⇒ hasDups_ (fst (split x)))
  (fun x ⇒ hasDups_ (fst (split x))) eq;

intuition.
  - apply PRF_A_wf.
  - unfold randomFunc_withDups.
  destruct (arrayLookup D_EqDec a b);
  fcf_well_formed.
  - fcf_well_formed.
  - subst.
  unfold randomFunc_withDups.
  case_eq (arrayLookup _ x2 a); intuition.

* fcf_irr_r.
  fcf_simp.
  fcf_spec_ret; simpl.

  remember (split x) as z.
  destruct z.
  simpl in *.
  trivial.
  simpl in *.
  remember (split x) as z.
  destruct z.
  simpl in *.
  destruct (in_dec (EqDec dec D_EqDec a l0); intuition.
  discriminate.
  rewrite notInArrayLookupNone in H.
  discriminate.
  intuition.
  rewrite unzip_eq_split in H3.
  remember (split x) as z.
  destruct z.
  pairInv.
  simpl in *.
  intuition.
  simpl in *.
```
No sense of hierarchy, importance, narrative. Where's the intuition?
Written by computers, for computers
The role of the human is not to understand, but to trust.
Text is a double-edged sword.
Powerful ways to manipulate and search text...
...but not pictures.
forall (a b c : object) {p:c -> a} {q:c -> b} : denote (parse [ "p c q " ; "+---------o--------+ " ; " | | " ; " | factr | " ; " | | " ; " v prjL v prjR v " ; " o<---------o--------->o " ; " a bundle b " ] c a (bundle a b) b p (factor p q) q projL projR);
The house believes that Coq is an incredible tool for thought.
Programmers’ affordances: precise language, instant feedback (correctness checking, REPL)
“You don’t have to know all the rules.”
Our only hope.
The house believes that Coq is a terrible tool for thought.
By computers, for computers.
Destroys intuition.
Built on top of:

written notation
text editors
Gallina
Ltac
checker
IDE
How can we make proof assistants “intuition assistants”?
Incremental improvements
Visualize theorem dependency tree
“Explanatory,” human-readable proofs: diff proof states
Translate proofs into English or a better tactic language
Lemma double_div2: forall n, div2 (double n) = n.
intro n.
induction n.
reflexivity.
unfold double in *|-*.
simpl.
rewrite <- plus_n_Sm.
rewrite IHn.
reflexivity.
Qed.

Now, we give the same proof using the new declarative language:

Lemma double_div2: forall n, div2 (double n) = n.
proof.
let n:nat.
per induction on n.
suppose it is 0.
  reconsider thesis as (0=0).
  thus thesis.
suppose it is (S m) and Hrec:thesis for m.
  have (div2 (double (S m))
      = div2 (S (S (double m)))).
      =~ (S (div2 (double m))).
      thus =~ (S m) by Hrec.
end induction.
end proof.
Qed.
Different interfaces for different areas of math
One visual interface: Ancient Greek Geometry
Calculemus is necessary—but not sufficient!
Thanks!

Katherine Ye
@hypotext
Appendix
Example 3: program equivalence
Theorem if_swap_equiv : forall (b : bool) (x y : nat),
(if b then x else y) = (if (negb b) then y else x).

Proof.
Theorem if_swap_equiv : forall (b : bool) (x y : nat),
(if b then x else y) = (if (negb b) then y else x).
Proof.
Theorem if_swap_equiv : \(\forall (b : \text{bool}) \ (x \ y : \text{nat}),\>
(if b then x else y) = (if \neg b \ b then y else x).
Proof.
intros.
Theorem if_swap_equiv : \forall (b : bool) (x y : nat),
  (if b then x else y) = (if (\neg b) b then y else x).

Proof.
  intros.
  destruct b.
Theorem if_swap_equiv : forall (b : bool) (x y : nat),
   (if b then x else y) = (if (negb b) then y else x).
Proof.
  intros.
  destruct b.

  (* Case 1: b = true *)
Theorem \texttt{if\_swap\_equiv} : \forall (b : \texttt{bool}) (x y : \texttt{nat}),
\( (\text{if } b \text{ then } x \text{ else } y) = (\text{if } \neg b \text{ then } y \text{ else } x) \).

Proof.
\begin{itemize}
  \item \texttt{intros}.
  \item \texttt{destruct }b\texttt{.}
\end{itemize}

(* Case 1: \( b = \texttt{true} \) *)
\begin{itemize}
  \item \texttt{- simpl.}
\end{itemize}
Theorem if_swap_equiv : forall (b : bool) (x y : nat),
(if b then x else y) = (if (negb b) then y else x).

Proof.
  intros.
  destruct b.

  (* Case 1: b = true *)
  - simpl.
    reflexivity.
Theorem if_swap_equiv : forall (b : bool) (x y : nat),
(if b then x else y) = (if (negb b) then y else x).
Proof.
  intros.
  destruct b.

  (* Case 1: b = true *)
  - simpl.
    reflexivity.

  (* Case 2: b = false *)
Theorem if_swap_equiv : forall (b : bool) (x y : nat),
(if b then x else y) = (if (negb b) then y else x).

Proof.
  intros.
  destruct b.

  (* Case 1: b = true *)
  - simpl.
    reflexivity.

  (* Case 2: b = false *)
  - simpl.
Theorem if_swap_equiv : forall (b : bool) (x y : nat),
    (if b then x else y) = (if (negb b) then y else x).
Proof.
    intros.
    destruct b.

    (* Case 1: b = true *)
    - simpl.
      reflexivity.

    (* Case 2: b = false *)
    - simpl.
      reflexivity.

No more subgoals.
(dependent evars:)
U:%- *response* All L1 (Coq Response)
if_swap_equiv is defined
Theorem if_swap_equiv_fast :
   forall (b : bool) (x y : nat),
   (if b then x else y) = (if (negb b) then y else x).
Proof.
   destruct b; reflexivity.
Qed.