Proof assistants as a tool for thought

Katherine Ye
@hypotext

Tools for thought workshop, March '16

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.
```

$$0 + y = y$$

$$(1 + x') + y = 1 + (x' + y)$$

```
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.
```

“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.
```

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

```
U:%%-  *goals*
1      = S (S O)
2      : nat
```

“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.
```

By definition

```
Theorem add0_left_id :
forall (n : nat), plus O n = n.
Proof.
```

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

0 is a right identity

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

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

```
1 subgoals, subgoal 1 (ID 39)

  ============================
   forall n : nat, plus n 0 = n
```

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

```
1 subgoals, subgoal 1 (ID 39)

  ============================
   forall n : nat, plus n 0 = n
```

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

```
2 subgoals, subgoal 1 (ID 43)

  ============================
   plus 0 0 = 0

subgoal 2 (ID 46) is:
 plus (S n') 0 = S n'
```

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

  (* Base case: n = 0 *)
  -
```

```
1 focused subgoals (unfocused: 1)
, subgoal 1 (ID 43)

  ============================
   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.
```

```
1 subgoals, subgoal 1 (ID 46)

subgoal 1 (ID 46) is:
 plus (S n') 0 = S 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') *)
  -
```

```
1 focused subgoals (unfocused: 0)
, subgoal 1 (ID 46)

  n' : nat
  IHn' : plus n' 0 = n'
  ============================
   plus (S n') 0 = S 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' *)
```

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

  n' : nat
  IHn' : plus n' 0 = n'
  ============================
   S (plus n' 0) = S 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' *)
    rewrite -> IHn'.
```

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

  n' : nat
  IHn' : plus n' 0 = n'
  ============================
   S n' = S 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' *)
    rewrite -> IHn'.
    reflexivity.
Qed.
```

```
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' *)
    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)

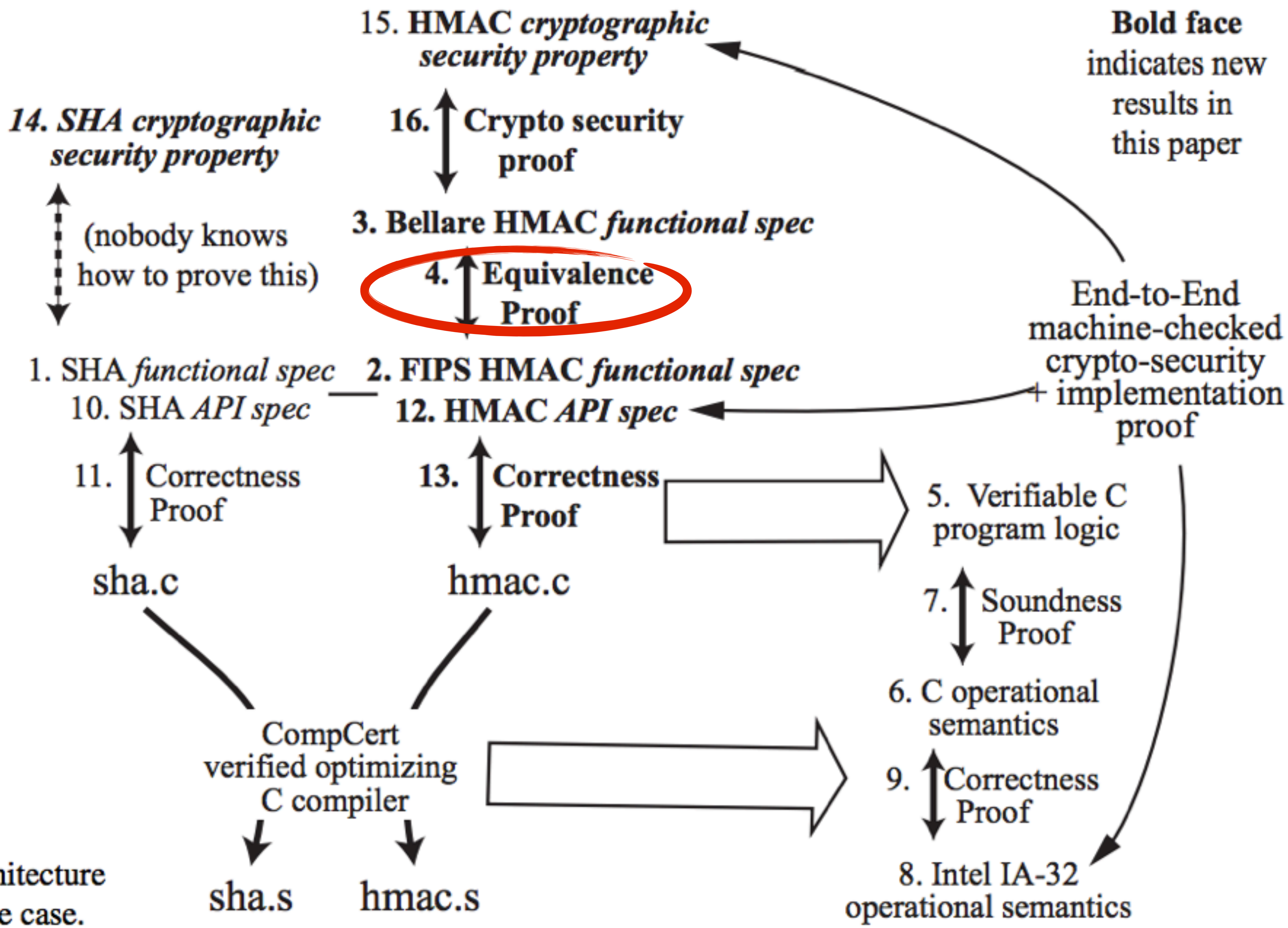


Figure 1: Architecture of our assurance case.

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...

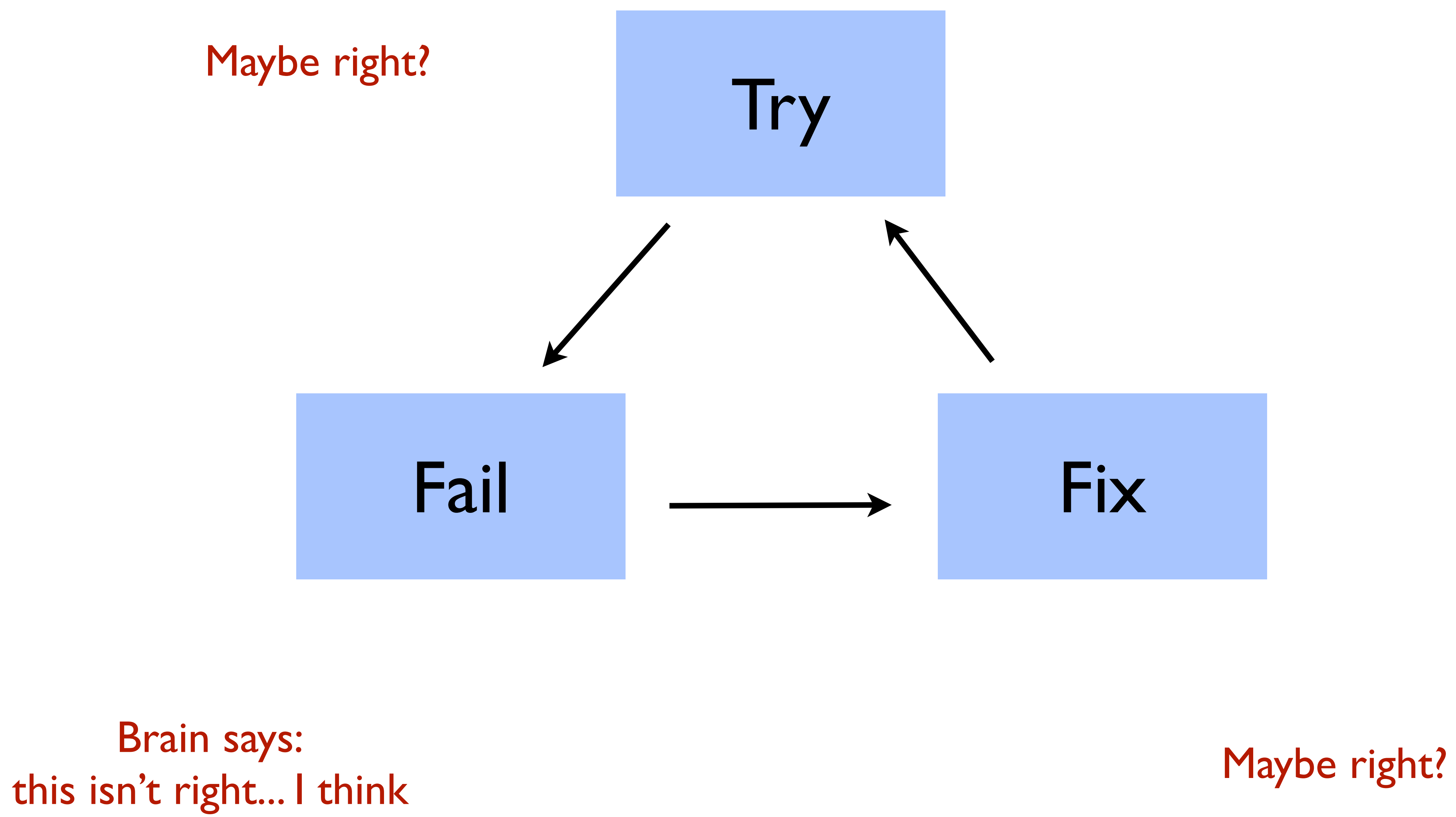
Proof state bookkeeping: assumptions, definitions, goals

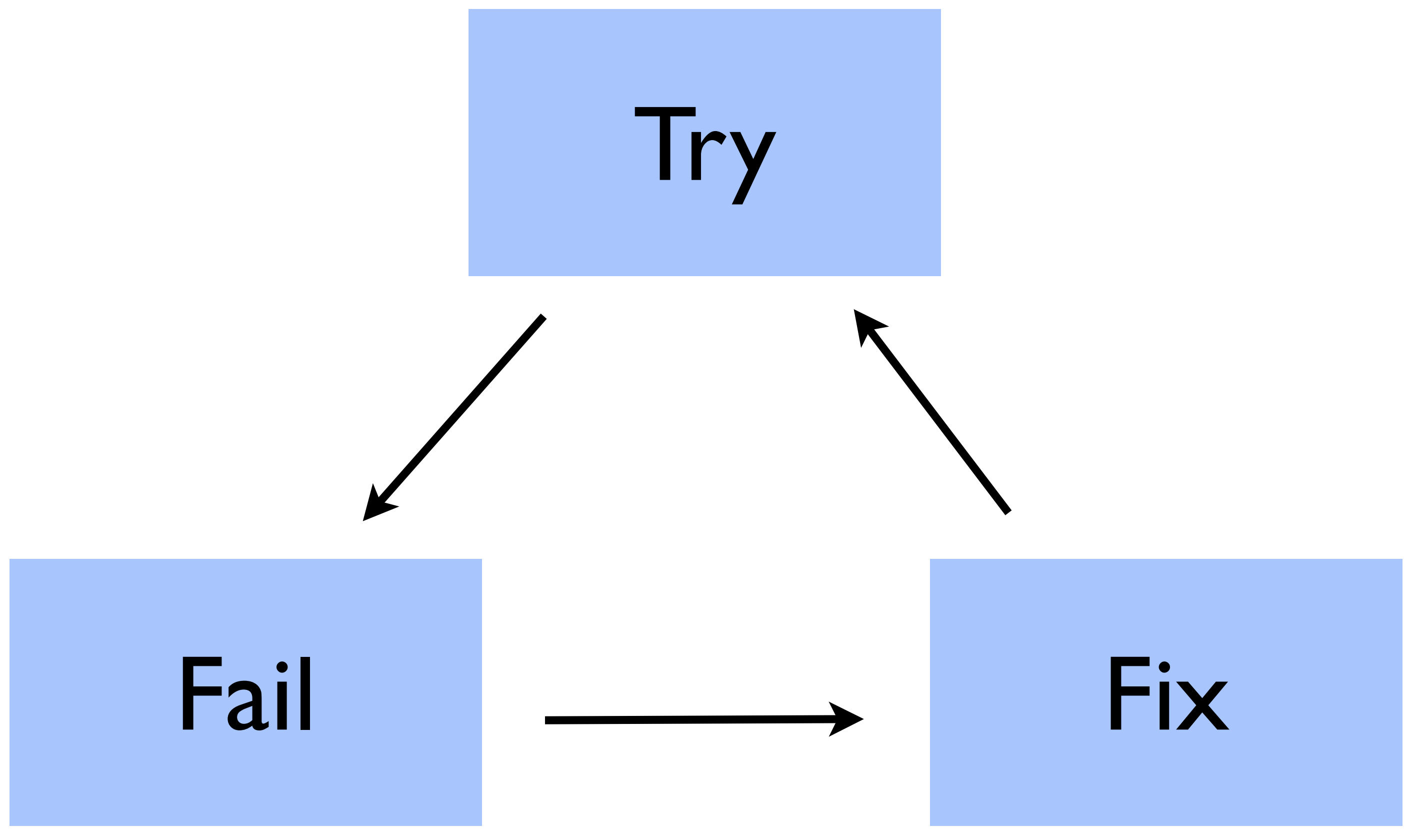
```
1 focused subgoals (unfocused: 0)
, subgoal 1 (ID 48)

  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 *)
```

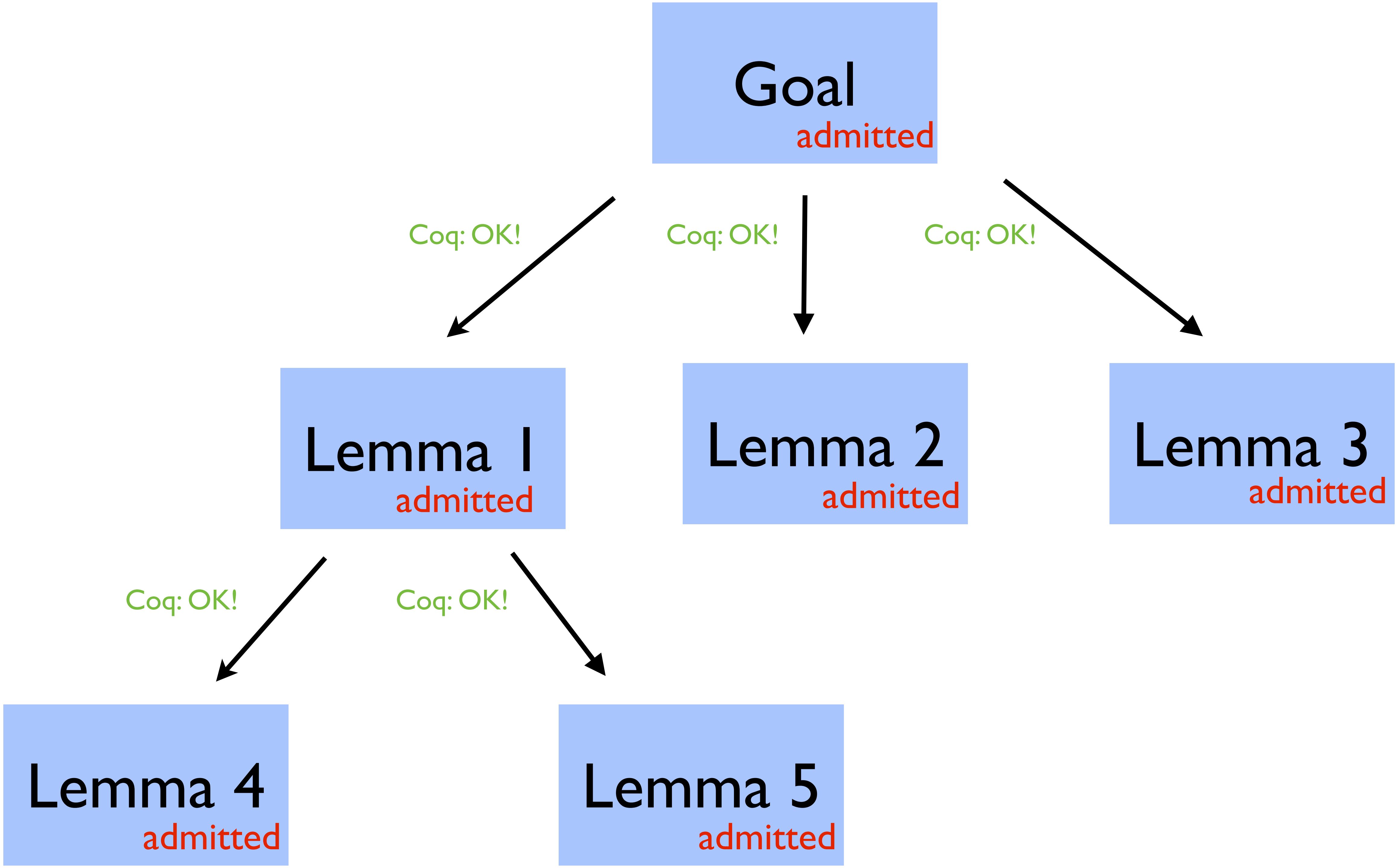
```
U:%%-  *goals*
1      = S (S 0)
2      : nat
```

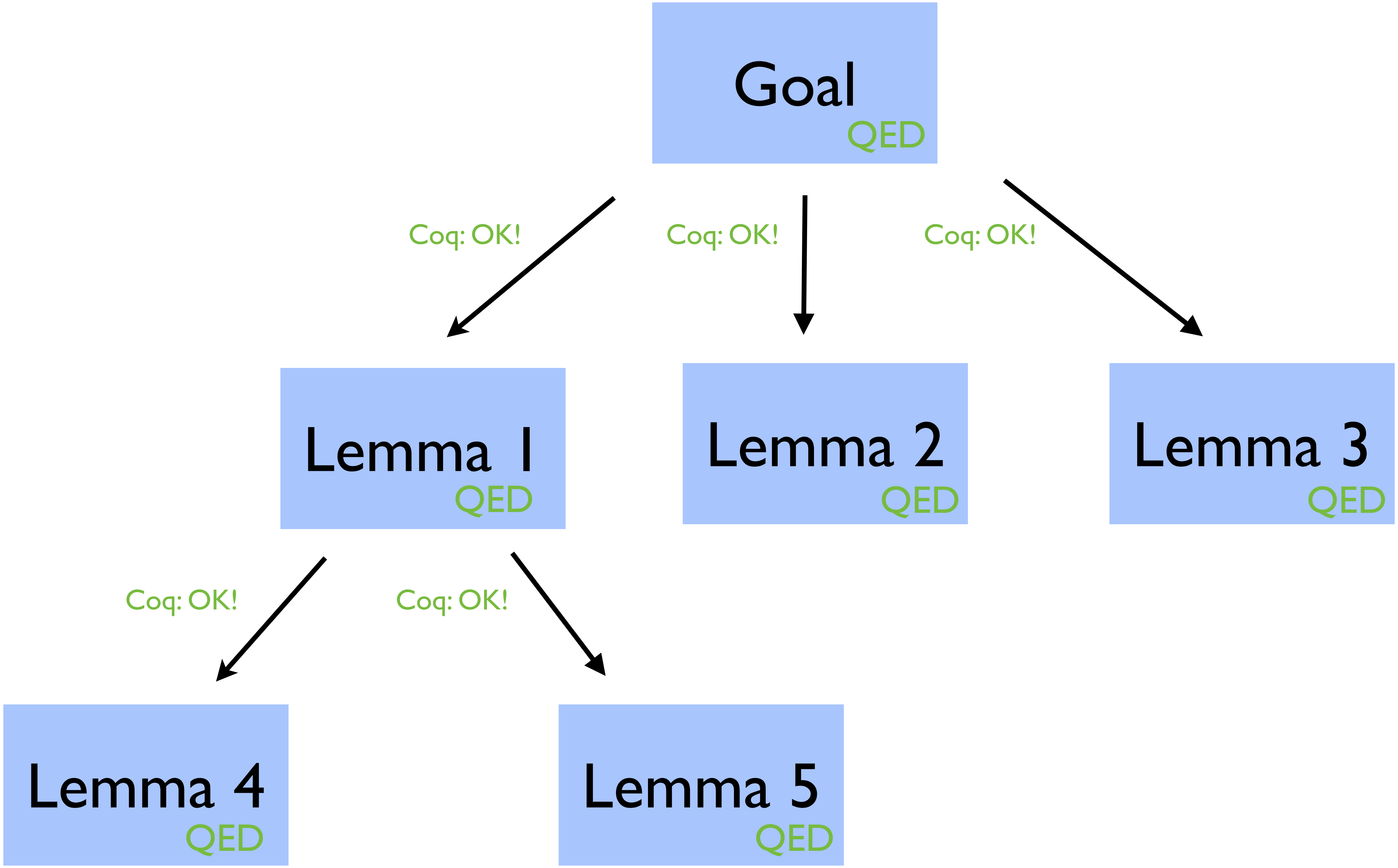

Maybe right?
Try
Fail
Fix
Brain says:
this isn't right... I think
Maybe right?



Coq says:
This is locally wrong!
Wrong type /
Can’t prove it /
Can’t use tactic /
Strange computation

Sketching with lemmas


Goal
admitted
Coq: OK!
Coq: OK!
Coq: OK!
Lemma 1
admitted
Lemma 2
admitted
Lemma 3
admitted
Coq: OK!
Coq: OK!
Lemma 4
admitted
Lemma 5
admitted



Goal
QED
Coq: OK!
Coq: OK!
Coq: OK!
Lemma 1
QED
Lemma 2
QED
Lemma 3
QED
Coq: OK!
Coq: OK!
Lemma 4
QED
Lemma 5
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.


2
3
4
$ 1500

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** ... **mechanically verifiable**.

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

2
3
4
1500

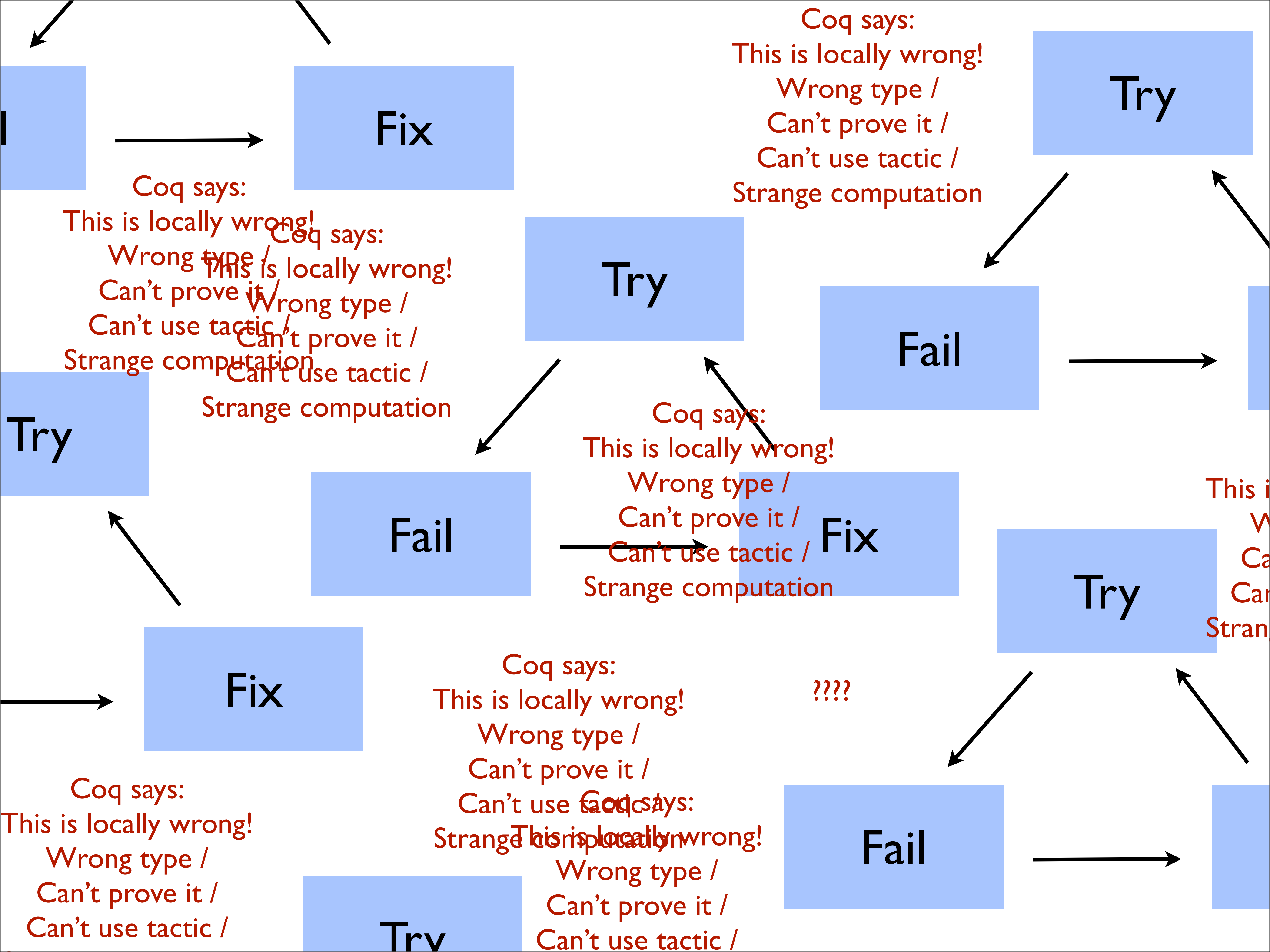
Work on paper first, then in computer :(

Coq blindfolds the intuition.
We’re not “Coq natives”!

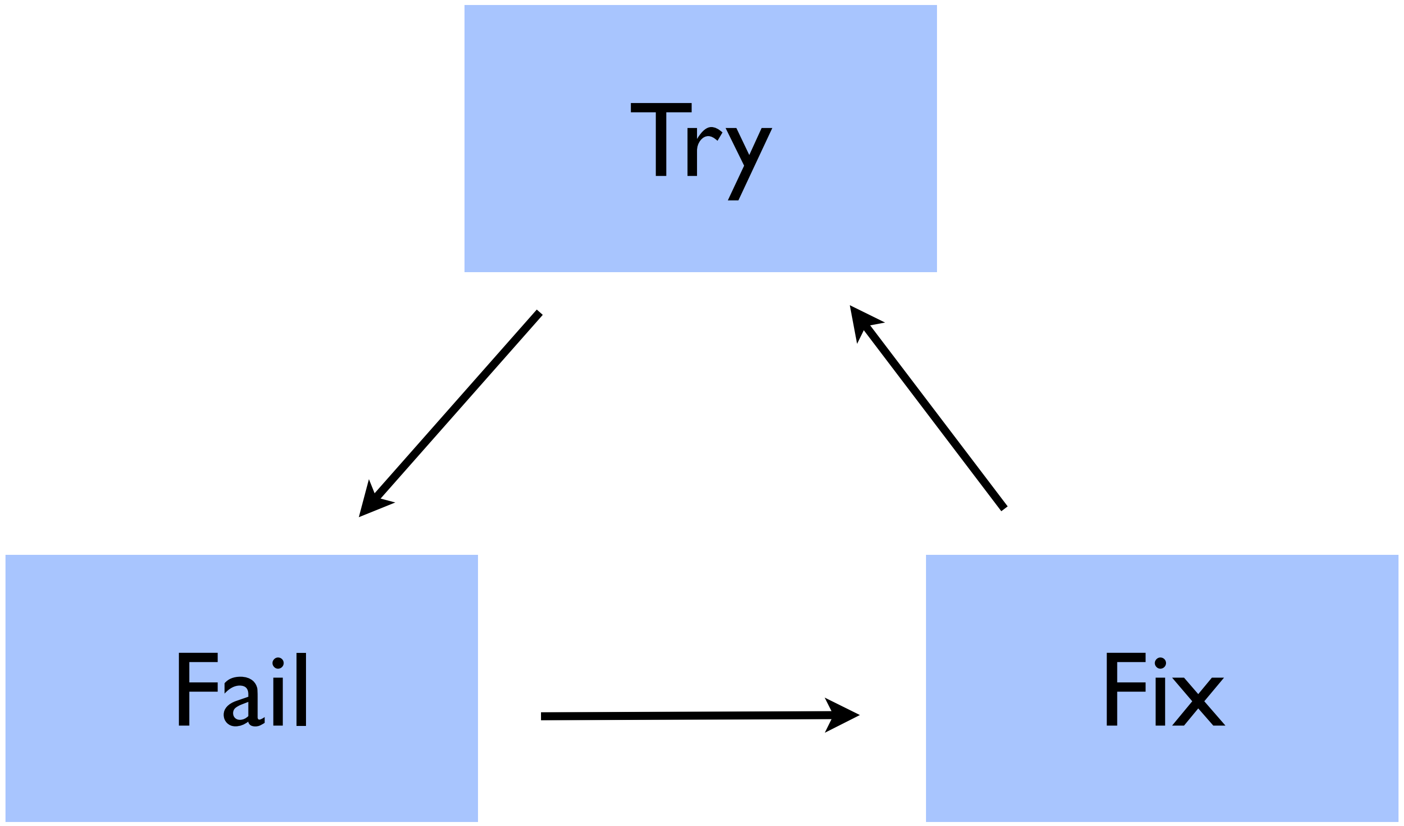
Built on top of:

written notation
text editors
Gallina
Ltac
checker
IDE

Started from the bottom...
now we're back.


Coq says:
This is locally wrong!
Wrong type /
Can't prove it /
Can't use tactic /
Strange computation
Try
Fix
Coq says:
This is locally wrong!
Wrong type /
Can't prove it /
Can't use tactic /
Strange computation
Coq says:
This is locally wrong!
Wrong type /
Can't prove it /
Can't use tactic /
Strange computation
Try
Fail
Try
Coq says:
This is locally wrong!
Wrong type /
Can't prove it /
Can't use tactic /
Strange computation
Fail
Fix
Try
Fix
Coq says:
This is locally wrong!
Wrong type /
Can't prove it /
Can't use tactic /
Strange computation
????
Coq says:
This is locally wrong!
Wrong type /
Can't prove it /
Can't use tactic /
Coq says:
This is locally wrong!
Wrong type /
Can't prove it /
Can't use tactic /
Fail

REPLs are alive
and make us reactive



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

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 <-$ { 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 *.
```

```
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 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 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.

```
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 *.
```

← 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 are all of these subgoals?

```
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?

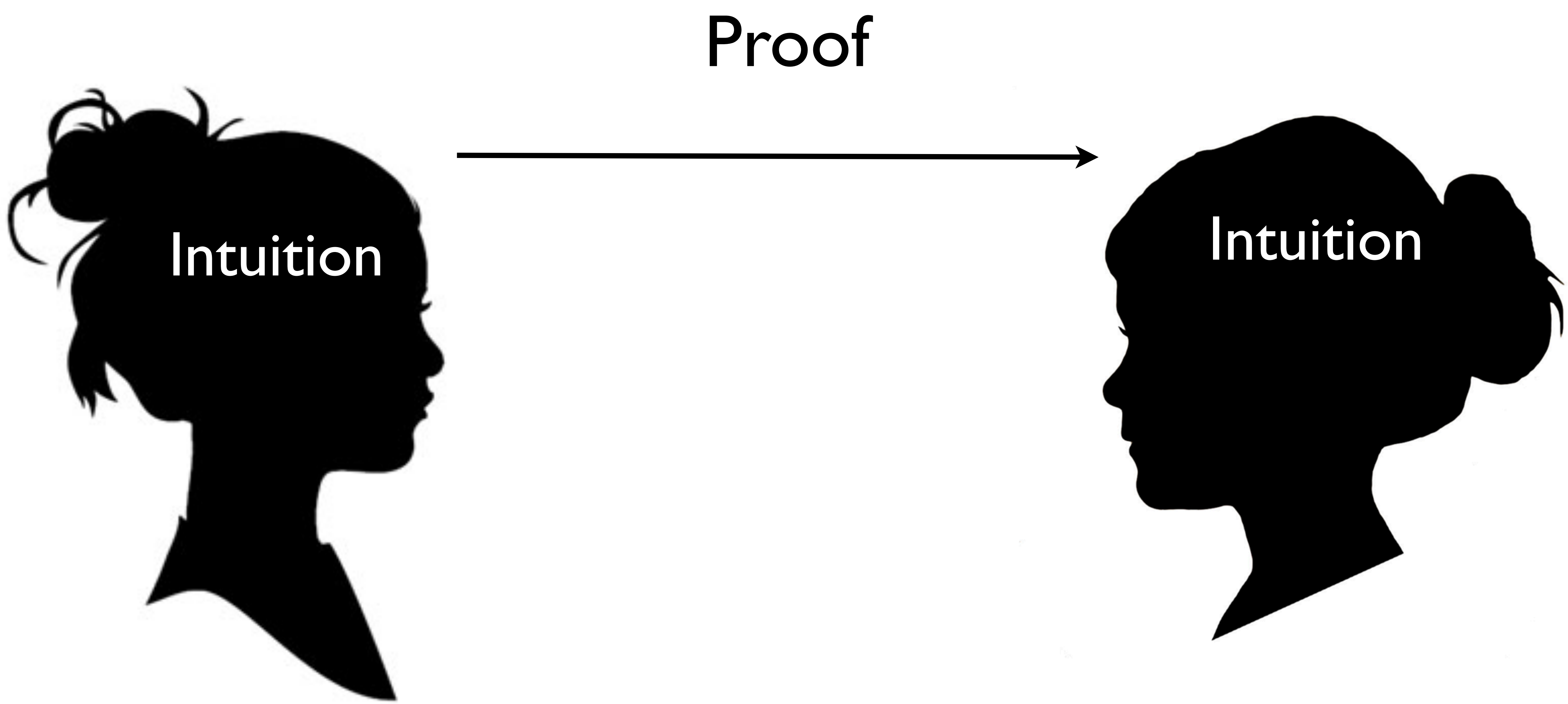
```
Theorem PRF_A_randomFunc_eq_until_bad :
  comp_spec
    (fun y1 y2 : bool * (list (D * Bvector eta)) =>
       hasDups _ (fst (split (snd y1))) =
       hasDups _ (fst (split (snd y2))) /\
       (hasDups _ (fst (split (snd y1))) = false ->
        snd y1 = snd y2 /\ fst y1 = fst y2))
    (PRF_A _ _ 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 *.
```

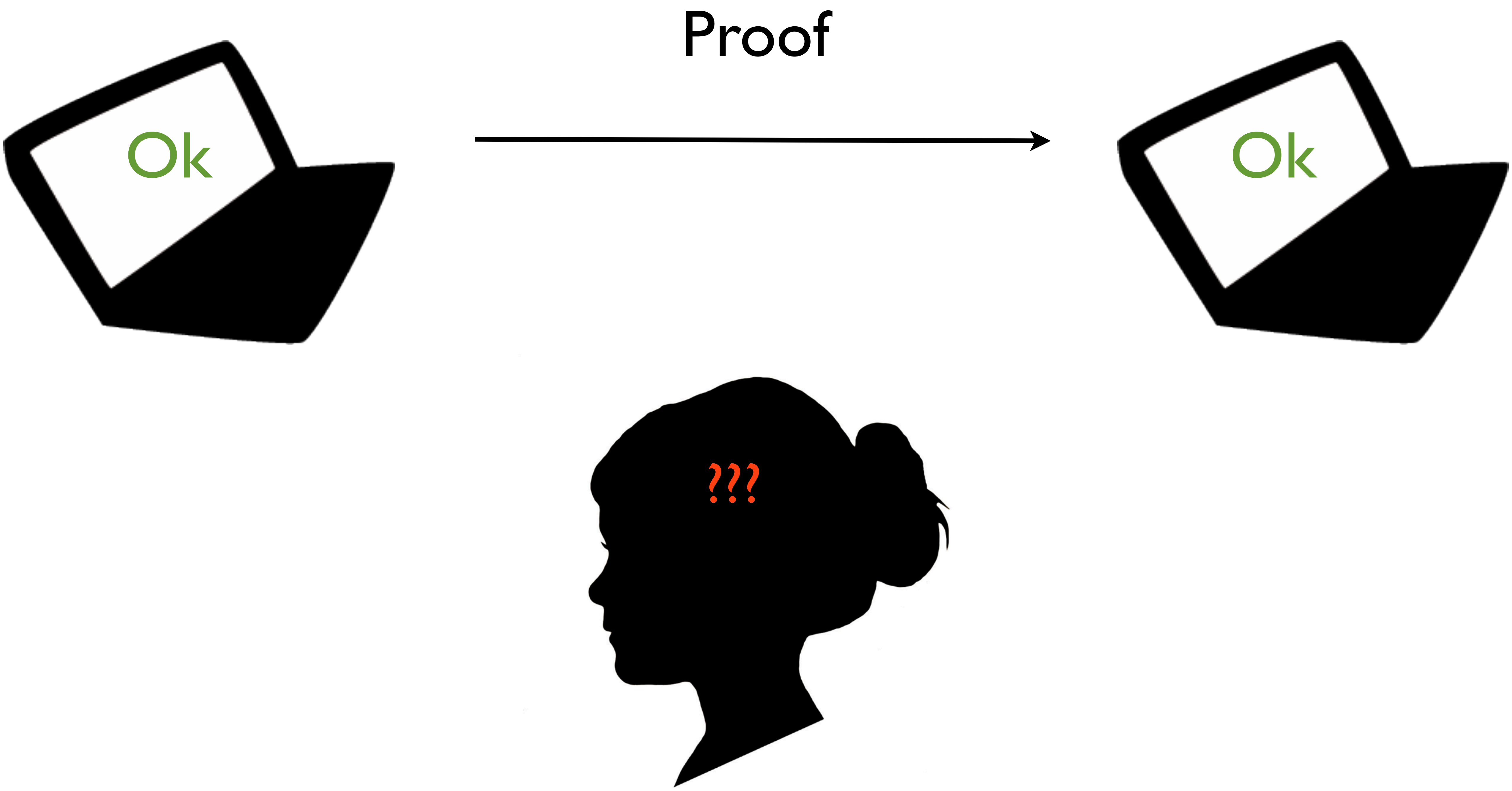
Wait, that proved the theorem???!!

```
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 *.
```

No sense of hierarchy, importance, narrative. Where’s the intuition?

Written by computers,
for computers


Proof
Intuition
Intuition


Proof
Ok
Ok
???

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.

Machine Checkable Pictorial Mathematics

```
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);
```

Proof by ASCII art

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.

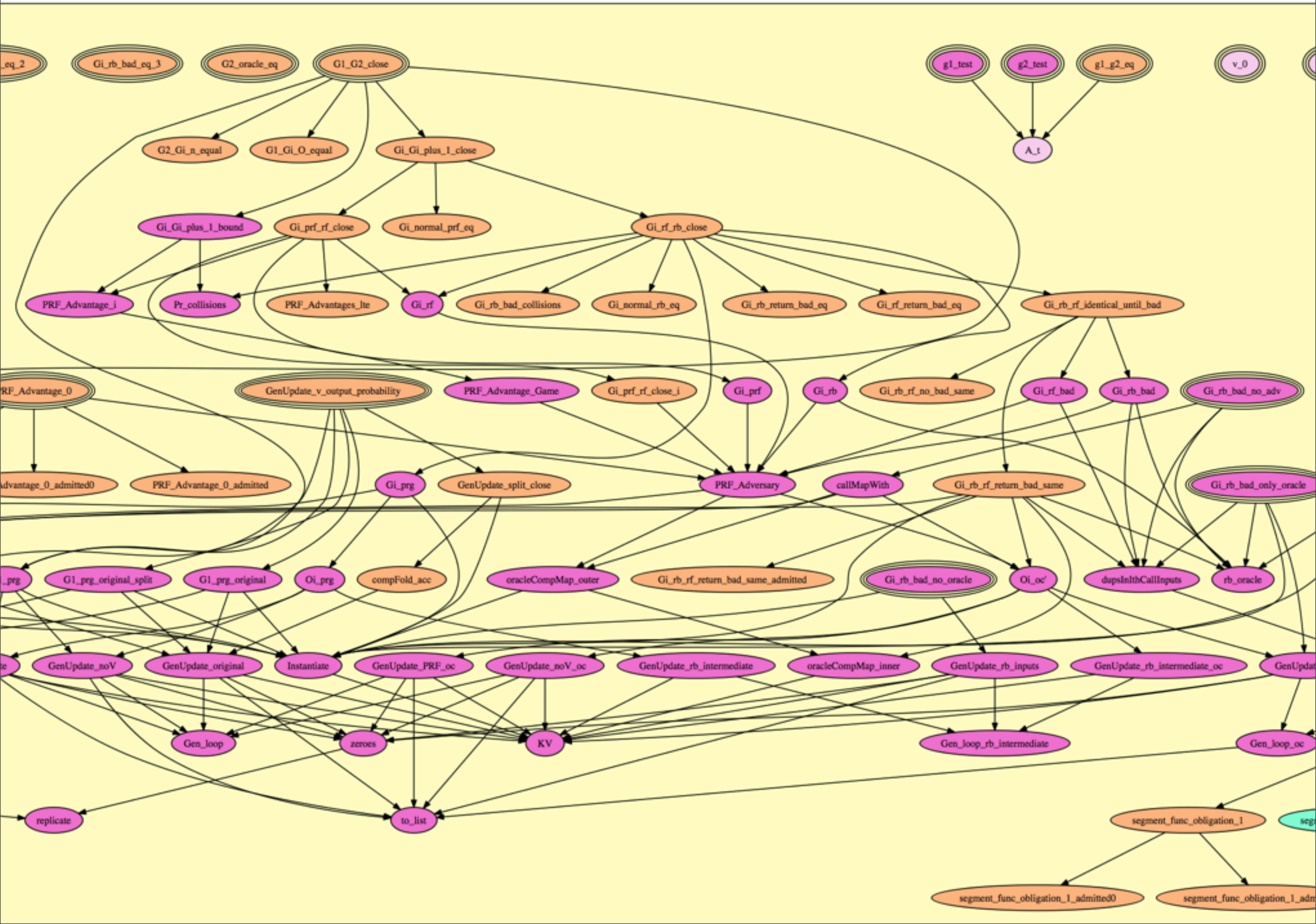
Built on top of:

written notation
text editors
Gallina
Ltac
checker
IDE

How can we make proof assistants "intuition assistants"?

Incremental improvements

Visualize theorem dependency tree



Gi_rb_bad_eq_3
G2_oracle_eq
G1_G2_close
g1_test
g2_test
g1_g2_eq
v_0
G2_Gi_n_equal
G1_Gi_O_equal
Gi_Gi_plus_1_close
A_t
Gi_Gi_plus_1_bound
Gi_prf_rf_close
Gi_normal_prf_eq
Gi_rf_rb_close
PRF_Advantage_i
Pr_collisions
PRF_Advantages_lte
Gi_rf
Gi_rb_bad_collisions
Gi_normal_rb_eq
Gi_rb_return_bad_eq
Gi_rf_return_bad_eq
Gi_rb_rf_identical_until_bad
GenUpdate_v_output_probability
PRF_Advantage_Game
Gi_prf_rf_close_i
Gi_prf
Gi_rb
Gi_rb_rf_no_bad_same
Gi_rf_bad
Gi_rb_bad
Gi_rb_bad_no_adv
PRF_Advantage_0_admitted
Gi_prg
GenUpdate_split_close
PRF_Adversary
callMapWith
Gi_rb_rf_return_bad_same
Gi_rb_bad_only_oracle
G1_prg_original_split
G1_prg_original
Oi_prg
compFold_acc
oracleCompMap_outer
Gi_rb_rf_return_bad_same_admitted
Gi_rb_bad_no_oracle
Oi_oc'
dupsInIthCallInputs
rb_oracle
GenUpdate_noV
GenUpdate_original
Instantiate
GenUpdate_PRF_oc
GenUpdate_noV_oc
GenUpdate_rb_intermediate
oracleCompMap_inner
GenUpdate_rb_inputs
GenUpdate_rb_intermediate_oc
Gen_loop
zeroes
KV
Gen_loop_rb_intermediate
Gen_loop_oc
replicate
to_list
segment_func_obligation_1
segment_func_obligation_1_admitted0

“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.
```

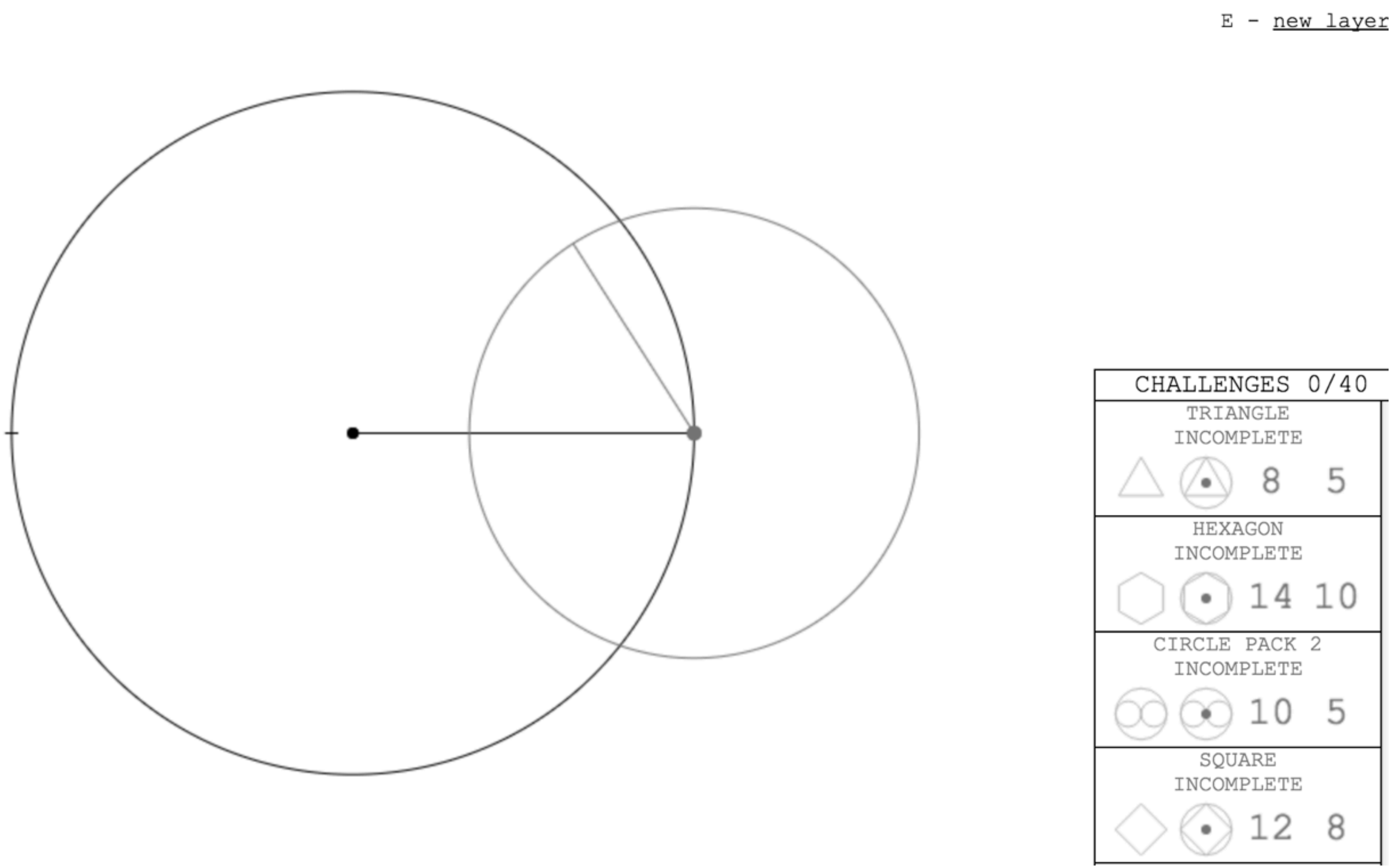
Now, we give the same proof using the new declarative language:

```
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.
```

A Declarative Language For The Coq Proof Assistant, Corbineau

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.
```

```
1 subgoals, subgoal 1 (ID 2)

  ============================
   forall (b : bool) (x y : nat),
   (if b then x else y) = (if negb b then y else x)
```

```
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.
```

```
1 subgoals, subgoal 1 (ID 5)

  b : bool
  x : nat
  y : nat
  ============================
   (if b then x else y) = (if negb b then y else x)
```

```
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.
```

```
2 subgoals, subgoal 1 (ID 8)

  x : nat
  y : nat
  ============================
   x = (if negb true then y else x)

subgoal 2 (ID 9) is:
 y = (if negb false then y else x)
```

```
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 *)
  -
```

```
1 focused subgoals (unfocused: 1)
, subgoal 1 (ID 8)

  x : nat
  y : nat
  ============================
   x = (if negb true then y else x)
```

```
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.
```

```
1 focused subgoals (unfocused: 1)
, subgoal 1 (ID 10)

  x : nat
  y : nat
  ============================
   x = x
```

```
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.
```

```
1 subgoals, subgoal 1 (ID 9)

subgoal 1 (ID 9) is:
 y = (if negb false then y else x)
```

```
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 *)
  -
```

```
1 focused subgoals (unfocused: 0)
, subgoal 1 (ID 9)

  x : nat
  y : nat
  ============================
   y = (if negb false then y else x)
```

```
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.
```

```
1 focused subgoals (unfocused: 0)
, subgoal 1 (ID 12)

  x : nat
  y : nat
  ============================
   y = y
```

```
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)
```

```
1   if_swap_equiv is defined

U:%%-  *response*      All L1      (Coq Response)
```

```
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.
```