An Engineer's Self-Taught Journey with Rocq Proof Assistant

Pierre-Emmanuel Wulfman

Graduate School of Mathematics, Nagoya University

Rocqshop 2025

Speaker Background

- M Eng in Information Science 2016
- 2019-2023: Compilation Engineer at Marigold (Tezos)
- Self-taught OCaml and Type Theory (Pierce, 2002 TPL)
- Colleagues use Coq to prove the compiler correct

Motivation

- 2022: Novel algorithm for red-black tree deletion
- Needed to prove correctness for publication
- Good opportunity to learn Rocq

Assumptions

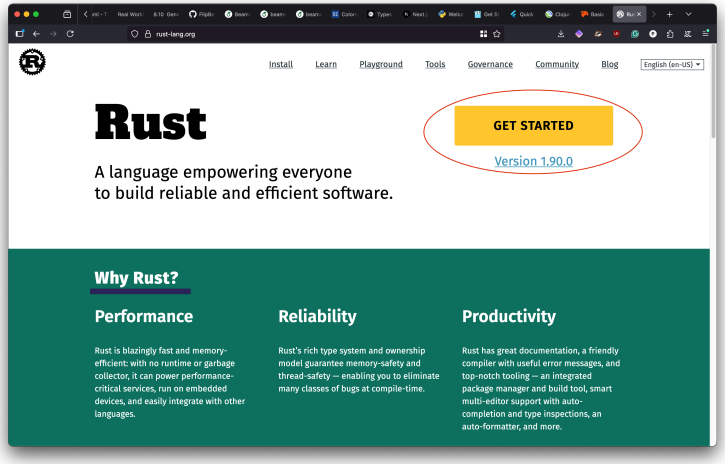
- The community want to attract 'standard' developers
  - Industry experienced programmers
  - Not necessarily familiar with functional programming
  - Not necessarily familiar with type theory
  - Mathematical proofs ?
- The online resources are intended to be used by such developers

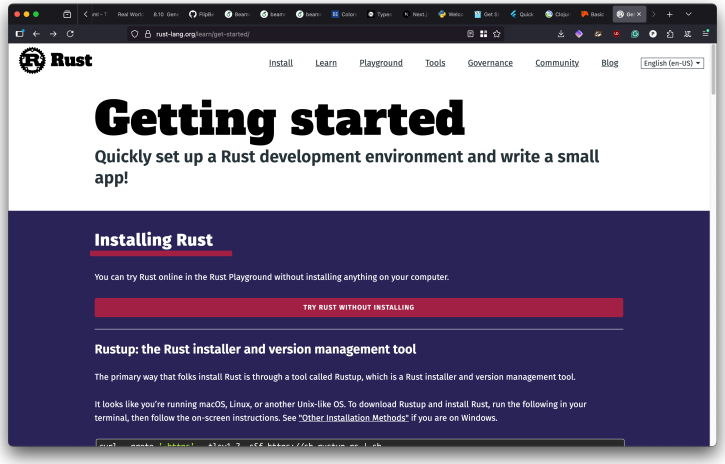
Culture difference

Developer	Researcher Culture
• Job is to build and ship	• Job is to learn and discover
• Goal is to make things work	• Goal is to formalize things
• Breadth of knowledge	• Depth of knowledge
• Learn by doing	• Learn by reading
• Have a goldfish attention span	• ...

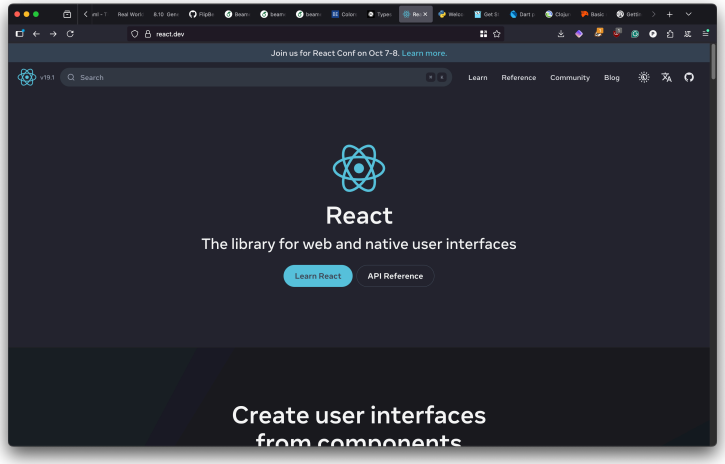
What Engineers Expect

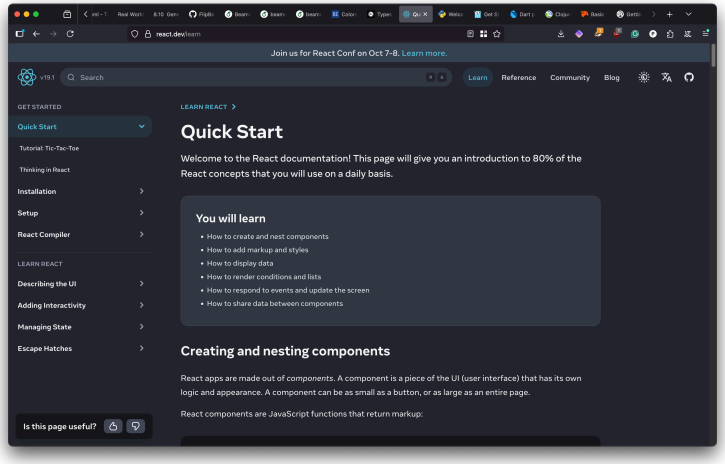
- Developer regularly learn new technologies
- In 2-3 clicks : minimal setup and examples (Hello World)
- Copy-paste code snippets to hack on.

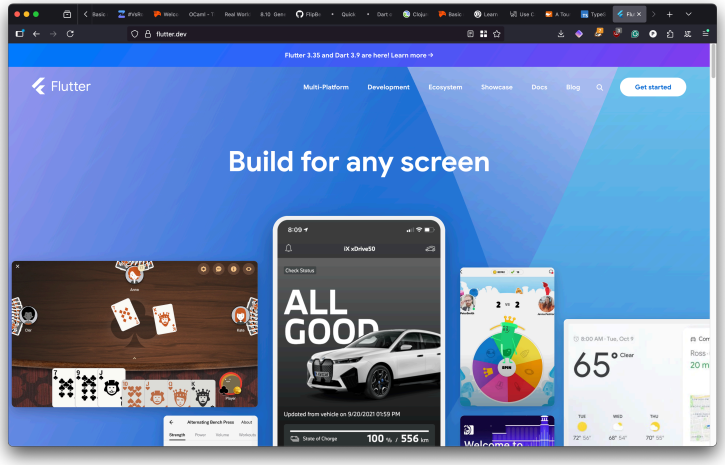


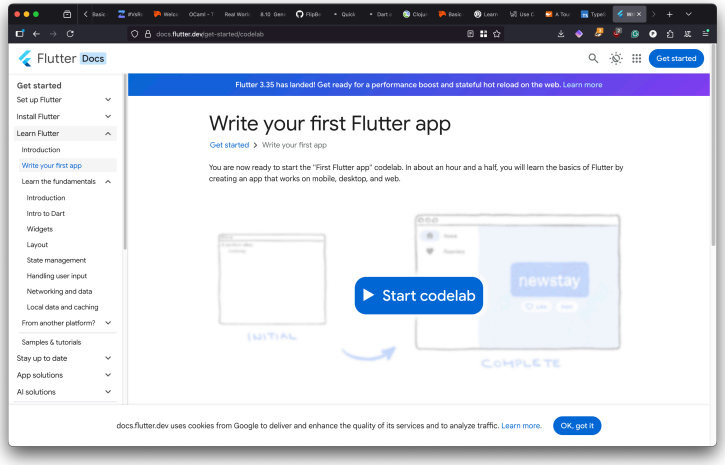


Other examples

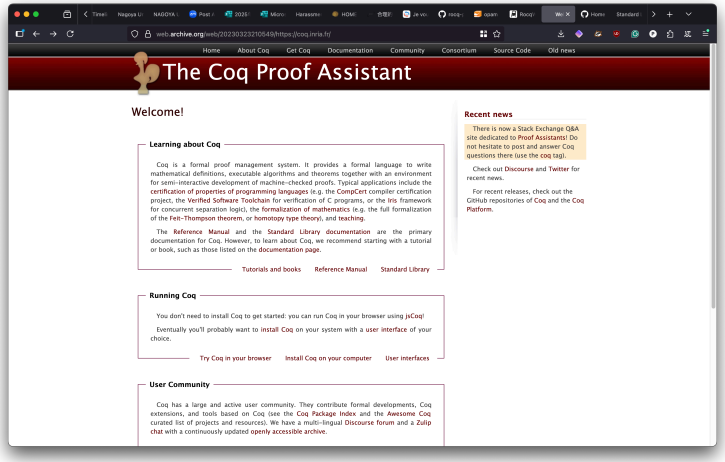


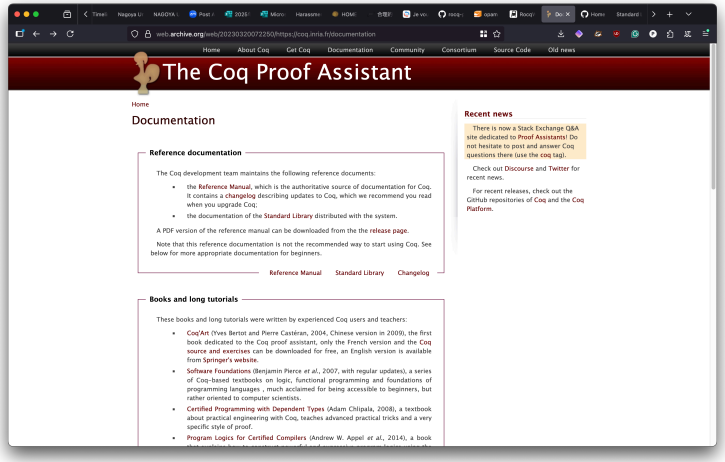


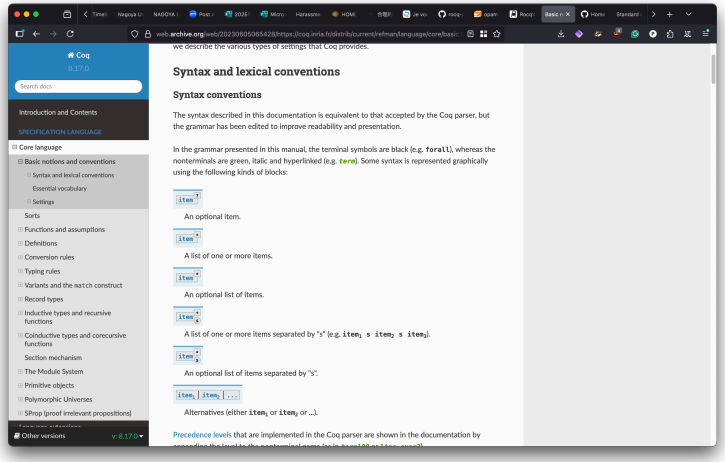


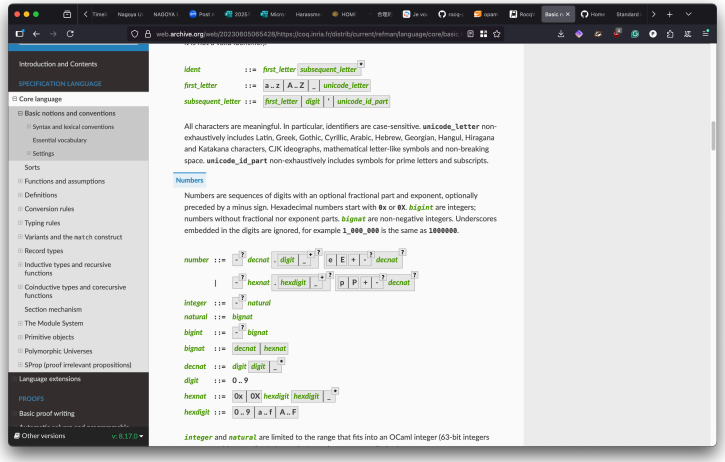


What I found for Coq

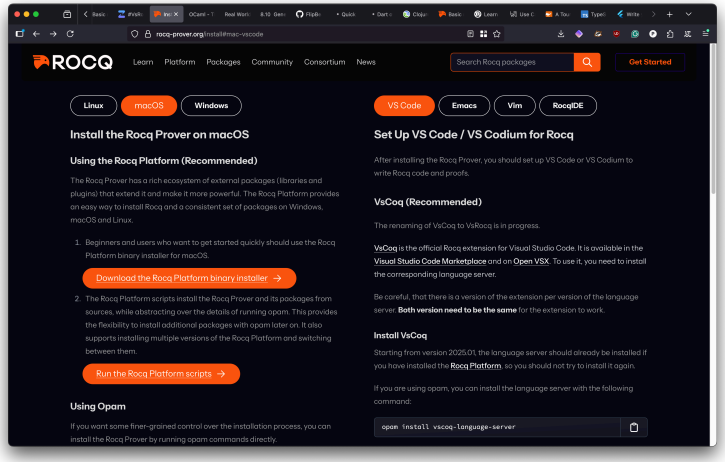


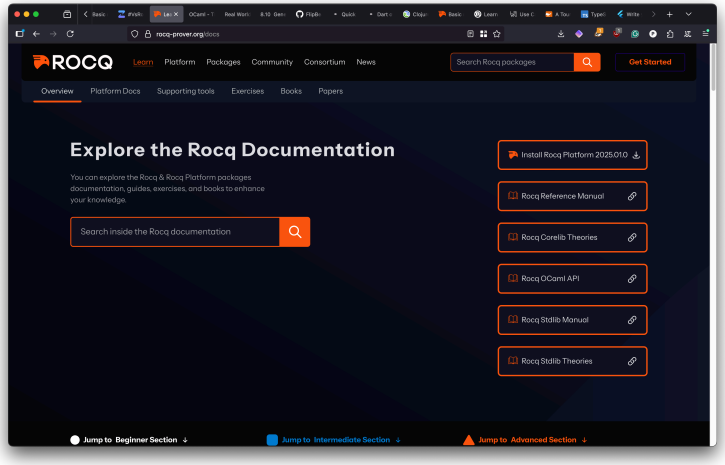


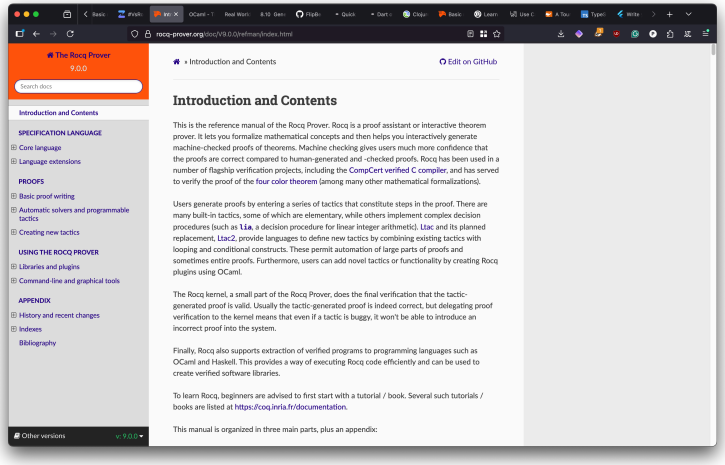




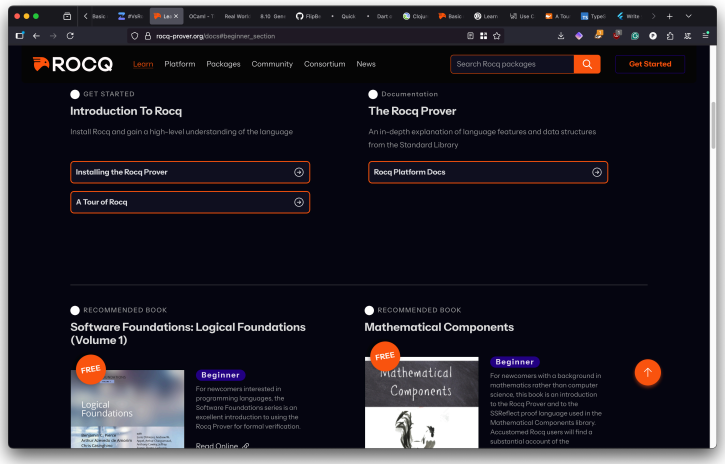
Rocq is better but...

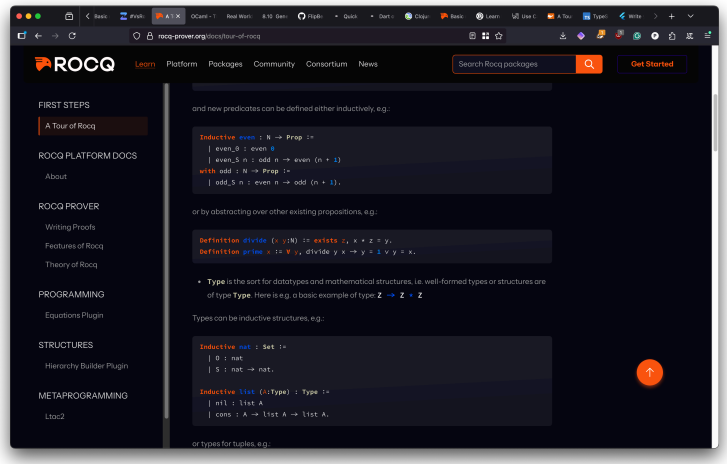




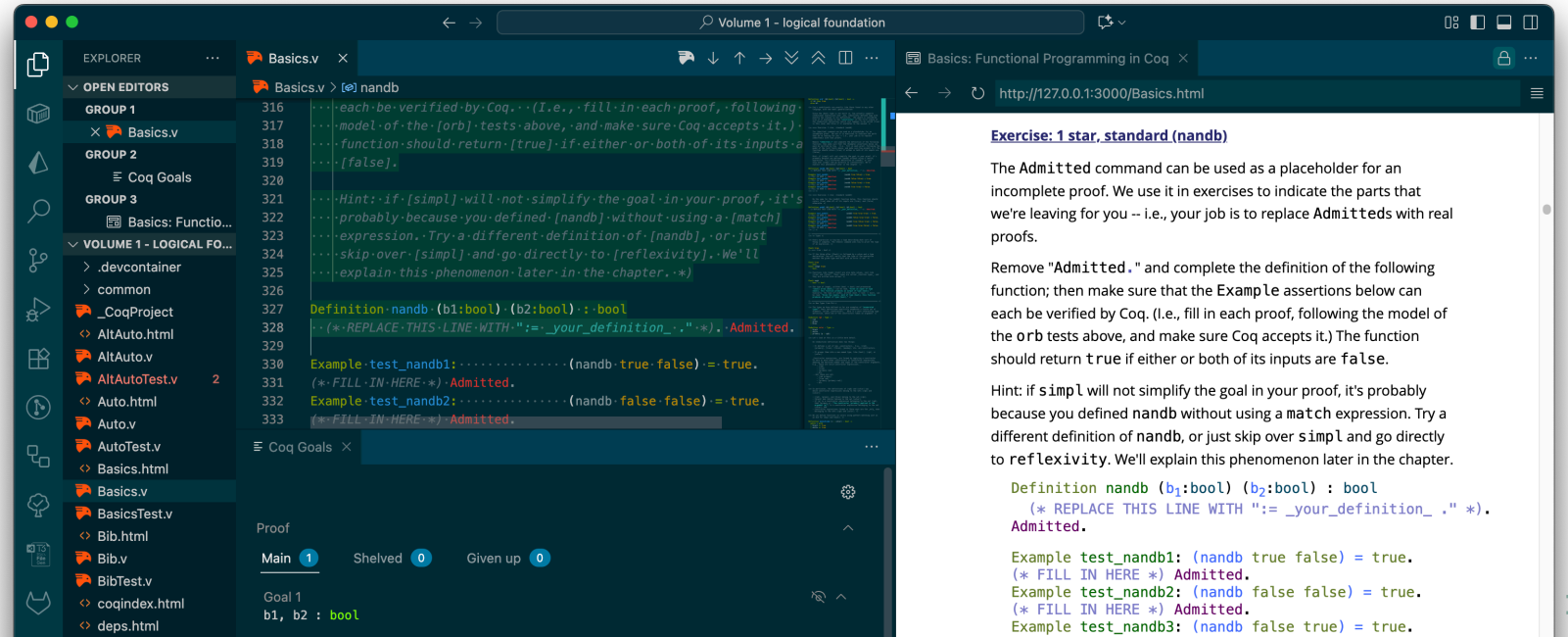


This should be more visible





Ended up using Software Foundations Vol. 1



Case Study: Red-Black Tree Deletion

what ?

Well-know data structure

why ?

Could not find an existing implem or proof

Seems simple

- Fairly simple structure

```
type colour = Red | Black

type 'a t = L | T of colour * 'a t * 'a * 'a t
💡
'a t
let empty = L      You, 3 years ago • add source file
```

- Fairly simple algorithm (about 100 lines)
- Fairly simple properties to express

```
(* BST properties *)
Fixpoint ForallT {X : Type} (P: X -> Prop) (t: tree X) : Prop :=
  match t with
  | Leaf => True
  | Node c l v r => P v /\ ForallT P l /\ ForallT P r
  end.

Inductive BST {X : Type} {cmp}: tree X -> Prop :=
| BST_Leaf : BST Leaf
| BST_Node : forall c l v r,
    ForallT (fun y => cmp y v = Lt) l ->
    ForallT (fun y => cmp y v = Gt) r ->
    BST l ->
    BST r ->
    BST (Node c l v r).
```

```
Inductive RedBlack {X} : tree X -> nat -> Prop :=
  | rb_leaf : RedBlack Leaf 0
  | rb_red_node : forall l r v n,
    RedBlack l n ->
    RedBlack r n ->
    IsBlack l ->
    IsBlack r ->
    RedBlack (Node Red l v r) n
  | rb_black_node : forall l r v n,
    RedBlack l n ->
    RedBlack r n ->
    RedBlack (Node Black l v r) (S n).
```

- Fairlty simple theorem

```
Theorem RB_delete: forall (t : tree T) (v : T),
  @RedBlacktree T cmp t -> @RedBlacktree T cmp (delete t v) .
```

Already made an informal explanation ! (proof ?)

(d). Then with a left rotation at the sibling node, we put (a), (b), (c), and (d) at the same height in the tree. By coloring the sibling black and (d) red, we restore the black property, but we may break the red property. For this reason, we have to call Okasaki's balance function on the right sibling restoring the red property at the sibling subtree and keeping the black property intact.

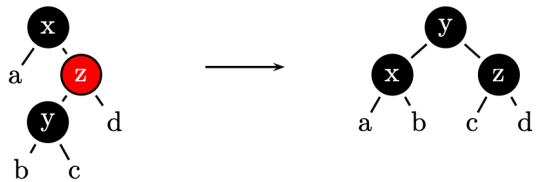


Figure 4: Case 1

The second case is when the sibling is black and has at least one red child (fig. 5). Still naming (a) the left node and (b), (c), (d) the red child's children and the other sibling's child, in this order. (a), (b), (c), and (d) have the same black height. Similarly, a left rotation at the right sibling (or a right rotation if the red child is at the right) and coloring the red child black fix the black property.

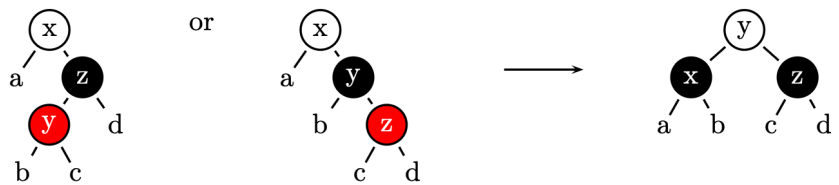
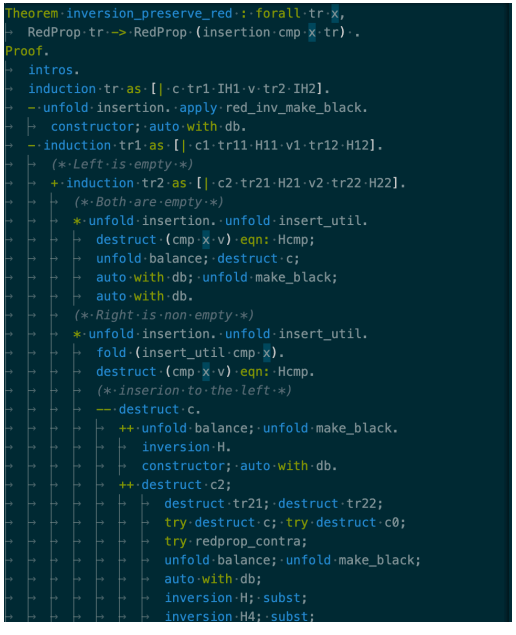


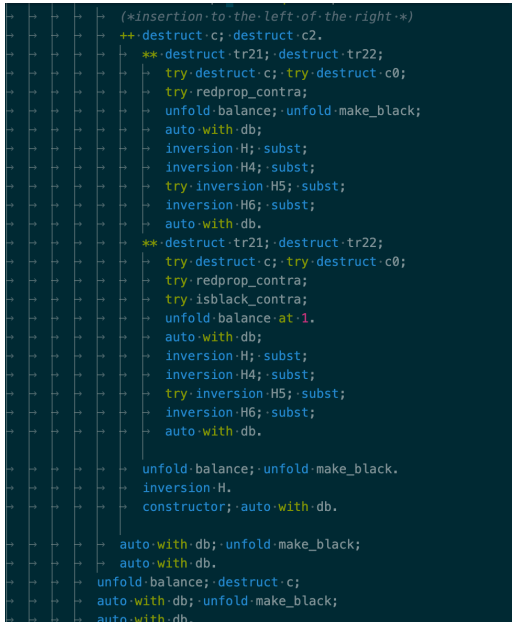
Figure 5: Case 2

Reality: Big chunky proof

Attempting to prove insertion first and failing



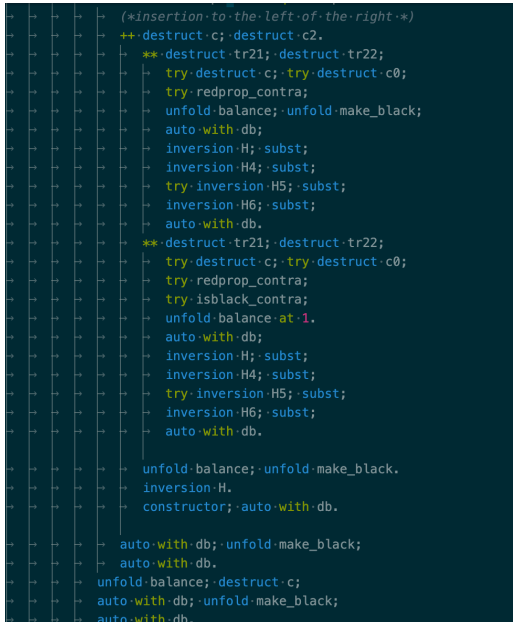
```
Theorem inversion_preserve_red : forall tr x,
  RedProp tr -> RedProp (insertion cmp x tr).
Proof.
  intros.
  induction tr as [| c tr1 IH1 v tr2 IH2].
  - unfold insertion. apply red_inv_make_black.
    constructor; auto with db.
  - induction tr1 as [| c1 tr11 H11 v1 tr12 H12].
    (* Left is empty *)
    + induction tr2 as [| c2 tr21 H21 v2 tr22 H22].
      (* Both are empty *)
      * unfold insertion. unfold insert_util.
        destruct (cmp x v) eqn: Hcmp;
        unfold balance; destruct c;
        auto with db; unfold make_black;
        auto with db.
      (* Right is non empty *)
      * unfold insertion. unfold insert_util.
        fold (insert_util cmp x).
        destruct (cmp x v) eqn: Hcmp.
        (* inserion to the left *)
        -- destruct c.
           ++ unfold balance; unfold make_black.
              inversion H.
              constructor; auto with db.
           ++ destruct c2;
              destruct tr21; destruct tr22;
              try destruct c; try destruct c0;
              try redprop_contra;
              unfold balance; unfold make_black;
              auto with db;
              inversion H; subst;
              inversion H4; subst;
```



```
(*insertion to the left of the right *)
++ destruct c; destruct c2.
   ** destruct tr21; destruct tr22;
      try destruct c; try destruct c0;
      try redprop_contra;
      unfold balance; unfold make_black;
      auto with db;
      inversion H; subst;
      inversion H4; subst;
      try inversion H5; subst;
      inversion H6; subst;
      auto with db.
   ** destruct tr21; destruct tr22;
      try destruct c; try destruct c0;
      try redprop_contra;
      try isblack_contra;
      unfold balance at 1.
      auto with db;
      inversion H; subst;
      inversion H4; subst;
      try inversion H5; subst;
      inversion H6; subst;
      auto with db.

   unfold balance; unfold make_black.
   inversion H.
   constructor; auto with db.

auto with db; unfold make_black;
auto with db.
unfold balance; destruct c;
auto with db; unfold make_black;
auto with db.
```



```
(*insertion to the left of the right *)
++ destruct c; destruct c2.
   ** destruct tr21; destruct tr22;
      try destruct c; try destruct c0;
      try redprop_contra;
      unfold balance; unfold make_black;
      auto with db;
      inversion H; subst;
      inversion H4; subst;
      try inversion H5; subst;
      inversion H6; subst;
      auto with db.
   ** destruct tr21; destruct tr22;
      try destruct c; try destruct c0;
      try redprop_contra;
      try isblack_contra;
      unfold balance at 1.
      auto with db;
      inversion H; subst;
      inversion H4; subst;
      try inversion H5; subst;
      inversion H6; subst;
      auto with db.

   unfold balance; unfold make_black.
   inversion H.
   constructor; auto with db.

auto with db; unfold make_black;
auto with db.
unfold balance; destruct c;
auto with db; unfold make_black;
auto with db.
```

Read Soft Found 3. and retry

intermediary data structures

```
Inductive RRBlack {X} : tree X -> nat -> Prop :=
  | rr_black_node : forall l r v n,
      RedBlack l n ->
      RedBlack r n ->
      RRBlack (Node Red l v r) n
  | rr_black_black_node : forall l r v n,
      RedBlack l n ->
      RedBlack r n ->
      RRBlack (Node Black l v r) (S n).
```

use of ltac(1) for automation

```
Ltac prove_RB :=
  repeat
    match goal with
    | |- _ (match ?x with Eq => _ | Lt => _ | Gt => _ end) _ => (
    | |- RRBlack _ _ => constructor
    | |- RedBlack _ _ => constructor
    | |- _ (match ?c with Red => _ | Black => _ end) _ => destru(
    | |- _ (match ?t with Leaf => _ | Node _ _ _ _ => _ end) _ =>
    | H: RRBlack Leaf _ |- _ => inv H
    | H: _ (Node _ _ _ _) _ |- _ => inv H
    | H: RedBlack Leaf _ |- _ => inv H
    | H: IsBlack (Node Red _ _ _) |- _ => inv H
    end;
  auto with db; try lia.      You, 2 months ago • Uncommitted cha
```

Pain points

- Proof to long ? split in smaller lemmas
- Lose context ? add invariants
- With one are correct ?

```
Theorem BST_ins : forall (t : tree T) (v : T) (x : T),
  @BST T cmp t -> (@BST T cmp (@insert_util T cmp v t)
  /\ (cmp v x = Lt ->
    ForallT (fun y : T => cmp y x = Lt) t ->
    ForallT (fun y: T => cmp y x = Lt) (@insert_util T cmp v t)
    )
  /\ (
    cmp v x = Gt ->
    ForallT (fun y : T => cmp y x = Gt) t ->
    ForallT (fun y: T => cmp y x = Gt) (@insert_util T cmp v t)
    )
  ).
```

Pain points

- What to do for polymorphic types ?
- What to do for constraints on polymorphic types ?
- Why does all my proofs start failing after a small change ?

Catch All tactics

```
Ltac prove_BST_del :=      You, 2 months ago • Uncommitted changes
  repeat
    match goal with
    | H : ForallT _ Leaf |- _ => inv H
    | H : ForallT _ (Node _ _ _ _) |- _ => inv H
    | H : BST Leaf |- _ => inv H
    | H : BST (Node _ _ _ _) |- _ => inv H
    | H : cmp ?x ?y = Lt |- cmp ?y ?x = Gt => apply inv_order
    | H : cmp ?x ?y = Gt |- cmp ?y ?x = Lt => apply inv_order
    | H : _ /\ _ |- _ => destruct H
    | |- BST (fst (match ?t with Leaf => _ | Node _ _ _ _ => _ end)) => destruct t
    | |- BST (fst (match ?c with Red => _ | Black => _ end)) => destruct c
    | |- BST (fst (if ?b then _ else _)) => destruct b
    | |- BST (fst (_, _)) => unfold fst
    | |- BST (balance _) => apply BST_balance
    | |- BST (redden _) => apply BST_redden
    | |- BST (Node _ _ _ _) => constructor
    | |- ForallT _ (fst (match ?t with Leaf => _ | Node _ _ _ _ => _ end)) => destruct t
    | |- ForallT _ (fst (match ?c with Red => _ | Black => _ end)) => destruct c
    | |- ForallT _ (fst (if ?b then _ else _)) => destruct b
    | |- ForallT _ (fst (_, _)) => unfold fst
    | |- ForallT _ (balance _) => apply BST_balance
    | |- ForallT _ (redden _) => apply BST_redden
    | |- ForallT _ (Node _ _ _ _) => constructor
    | |- _ /\ _ => split
    | H: ForallT (fun _ : T => cmp _ ?x = Gt) ?t
      , GMP: cmp ?y ?x = Lt
      |- ForallT (fun _ : T => cmp _ ?y = Gt) ?t => apply (forall_cmp_trans_gt t x y)
    | H: cmp ?x ?y = Lt,
      GMP: cmp ?z ?y = Gt
      |- cmp ?x ?z = Lt => apply (order_trans x y z)
    | H: ForallT (fun _ : T => cmp _ ?x = Lt) ?t
```

Easy proof

```
Theorem BST_bal_right : forall x c l v (rb : tree T * bool),
  @BST T cmp (Node c l v (fst rb)) -> (@BST T cmp (fst (balance_right c l v rb))
  /\ (
    (ForallT (fun y : T => cmp y x = Lt) (Node c l v (fst rb)) ->
    ForallT (fun y: T => cmp y x = Lt) (fst(balance_right c l v rb)))
  /\ (ForallT (fun y : T => cmp y x = Gt) (Node c l v (fst rb)) ->
    ForallT (fun y: T => cmp y x = Gt) (fst(balance_right c l v rb)))
    )).
Proof.
  intros. destruct rb.
  repeat split; intros;
  unfold balance_right; prove_BST_del.
Qed.
```

What I would like to see

- Easy to find examples (Cookbook)
- High level libraries
- Definition vs Inductive vs Fixpoint vs ...
- Lemmas vs Theorem vs ...
- 'Easy refactoring tools'

Questions ?

Thank you !