

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

- Job is to build and ship
- Goal is to make things work
- Breadth of knowledge
- Learn by doing
- Have a goldfish attention span

Researcher Culture

- Job is to learn and discover
- Goal is to formalize things
- Depth of knowledge
- Learn by reading
- ...

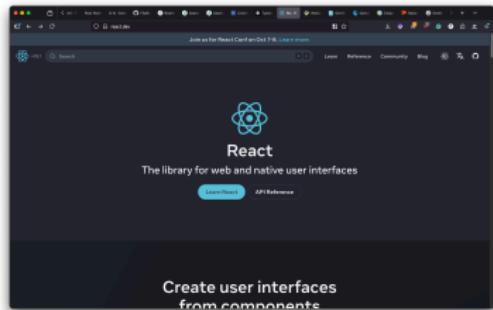
What Engineers Expect

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

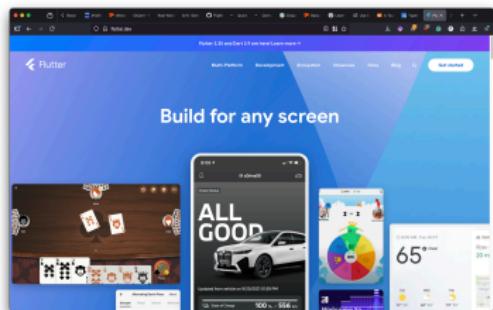
The screenshot shows the official Rust website. At the top, there's a navigation bar with links like 'Install', 'Learn', 'Playground', 'Tools', 'Governance', 'Community', and 'Help'. Below the navigation, the word 'Rust' is prominently displayed in a large, bold, black font. Underneath it, a sub-headline reads 'A language empowering everyone to build reliable and efficient software.' A large yellow 'GET STARTED' button is centered below this text, with a red oval highlighting it. Below the button, the text 'Version 1.92.0' is visible. The main content area has a dark green background and features three sections: 'Why Rust?' (with sub-sections 'Performance', 'Reliability', and 'Productivity'), 'The Rust Programming Language', and 'The Rustonomicon'. Each section contains brief descriptions and links to more information.

This screenshot shows a specific section of the Rust documentation titled 'Getting started'. The title is in large, bold, black letters. Below it, a sub-headline says 'Quickly set up a Rust development environment and write a small app!'. A prominent red button labeled 'TRY RUST ONLINE' is centered. Below this, another section titled 'Installing Rust' is shown with a sub-headline 'You can try Rust online in the Rust Playground without installing anything on your computer.' It includes a link 'TRY RUST ONLINE'. Further down, there's a section about 'Using the Rust installer and version management tool' with a note about 'Rustup' and instructions for Mac, Linux, and Windows users.

Other examples

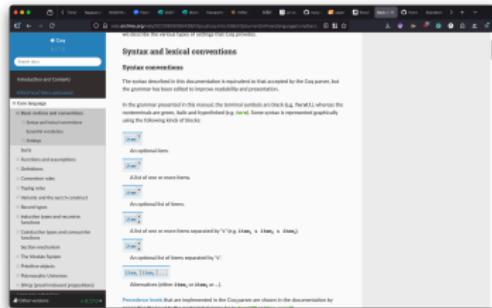
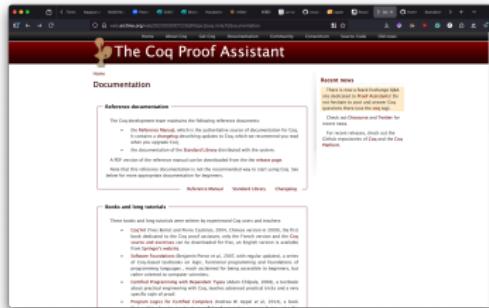
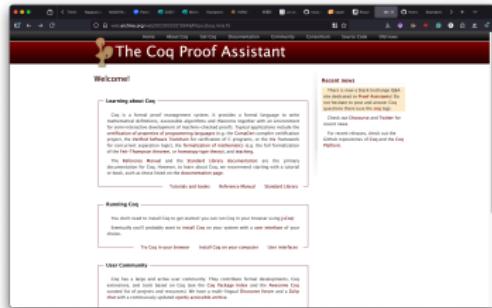


This screenshot displays the 'Quick Start' section of the React documentation. It includes a sidebar with navigation links like 'Get Started', 'Quick Start', 'Topics To Cover', 'Getting started', 'Installations', 'Setting', 'Press Committee', 'LICENSE REQUEST', 'Choosing the CR', 'Adding Ideas or Issues', 'Managing Status', and 'Filing Feedback'. The main content area is titled 'Quick Start' and describes the concepts learned here. It also includes a 'You will learn' section with bullet points about creating and nesting components, and a 'Creating and nesting components' section with a detailed explanation.

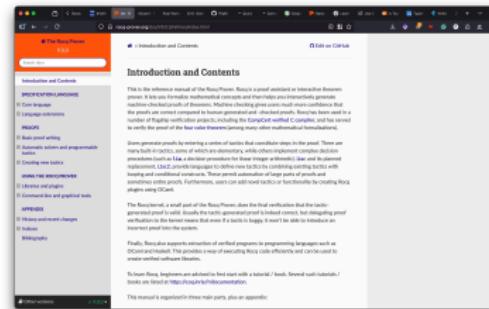
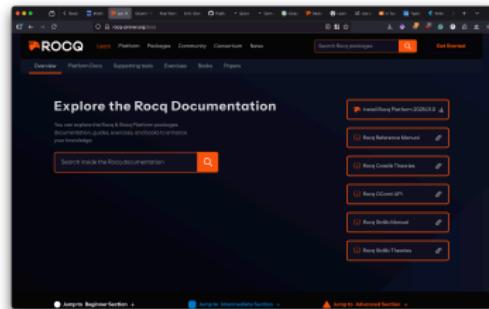
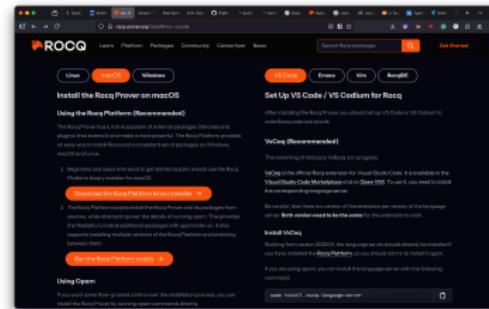


This screenshot shows the 'Get Started' page of the Flutter documentation. It features a sidebar with links for 'Get started', 'Set up Flutter', 'Install Flutter', 'Learn Flutter', and 'Flutter Devs'. The main content area is titled 'Write your first Flutter app' and includes a 'Start codelab' button. It also contains a note about the 'First Flutter app' codelab and a 'DEVELOP' button.

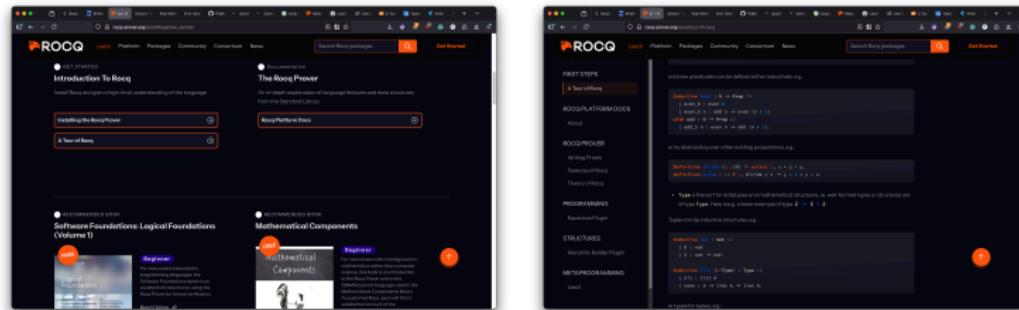
What I found for Coq



Rocq is better but...



This should be more visible



Ended up using Software Foundations Vol. 1

```
Basics.v (w) nandb
316 each-be-verified-by Coq... (I.e., fill-in-each-proof, following
317 model-of-the-(orb)-tests-above, and make-sure-Coq-accepts-it.)
318 function-should-return true if either or both of its inputs is
319 false.
320
321 Hint: if (simp1) will-not-simplify-the-goal-in-your-proof, it's
322 probably-because you-defined (nandb)-without-using-a-match-expression.
323 Try-a-different-definition-of (nandb), or just
324 skip-over (simp1) and-go-directly-to (reflexivity). We'll
325 explain-this-phenomenon-later-in-the-chapter. */
326
327 Definition nandb (b1:bool) (b2:bool) : bool
328 (* REPLACE THIS LINE WITH ":= _your_definition_." *). Admitted.
329
330 Example test_nandb1 : (nandb true false) = true.
331 (* FILL IN HERE *) Admitted.
332 Example test_nandb2 : (nandb false false) = true.
333 (* FILL IN HERE *) Admitted.
```

Exercise: 1 star, standard (nandb)

The `Admitted` command can be used as a placeholder for an incomplete proof. We use it in exercises to indicate the parts that we're leaving for you -- i.e., your job is to replace `Admitted`s with real proofs.

Remove "`Admitted.`" and complete the definition of the following function; then make sure that the `Example` assertions below can each be verified by Coq. (I.e., fill in each proof, following the model of the orb tests above, and make sure Coq accepts it.) The function should return `true` if either or both of its inputs are `false`.

Hint: if `simp1` will not simplify the goal in your proof, it's probably because you defined `nandb` without using a `match` expression. Try a different definition of `nandb`, or just skip over `simp1` and go directly to reflexivity. We'll explain this phenomenon later in the chapter.

```
Definition nandb (b1:bool) (b2:bool) : bool
(* REPLACE THIS LINE WITH ":= _your_definition_." *). Admitted.

Example test_nandb1 : (nandb true false) = true.
(* FILL IN HERE *) Admitted.
Example test_nandb2 : (nandb false false) = true.
(* FILL IN HERE *) Admitted.
Example test_nandb3 : (nandb false true) = true.
```

Case Study: Red-Black Tree Deletion

what ?

Well-known 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
    |
    'at
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.

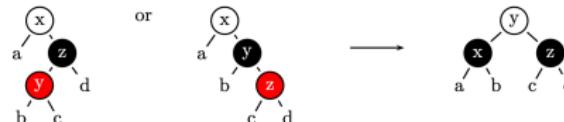


Figure 5: Case 2

Reality: Big chunky proof

Attempting to prove insertion first and failing

```

source /usr/local/proverus/lib/forall.tr.l
-RedProp -tr=>RedProp (insertion cmp & tr) .
Proof.
intros.
induction tr as [| c tr1 H1 v2 tr2 H2].
= unfold insertion, apply red_imv_make_black.
| constructor; auto with db.
- induction tr1 as [| c tr11 H11 v1 tr22 H21].
  + induction tr2 as [| c2 tr21 H12 v2 tr22 H21].
    + (* Left-is-not-empty *)
      unfold insertion, unfold insert_util.
      destruct (cmp v1 v2) eqn:Hump.
      unfold balance; destruct c;
      auto with db; unfold make_black;
      auto with db.
    + (* Right-is-not-empty *)
      unfold insertion, unfold insert_util.
      fold (insert_util cmp #).
      destruct (cmp v2 v1) eqn:Hump.
      (* Left-is-not-empty *)
      destruct c.
      + (* Left-is-empty *)
        unfold balance; unfold make_black.
        inversion H1.
        constructor; auto with db.
      + (* Left-is-not-empty *)
        destruct c2;
        destruct tr21; destruct tr22;
        try destruct c2 try destruct d0;
        try redprop_contr.
        unfold balance; unfold make_black;
        auto with db;
        inversion H1 subst;
        inversion H1 subst;

```

```

  <constructor> (to the left of :m: the right >
++ construct c; destruct c2,
++ ~destruct tr2l; destruct tr2r;
++ try destruct c; try destruct c#;
++ unfold balance; unfold make_black;
++ auto with db;
++ inversion Hj; substi;
++ inversion H4; substi;
++ try inversion H5; substi;
++ inversion H6; substi;
++ auto with db;
++ ~destruct tr2l; ~destruct tr2r;
++ try destruct c; try destruct c#;
++ try redprop_contr;
++ try isblack_contr;
++ unfold balance at 1.
++ auto with db;
++ inversion Hj; substi;
++ inversion H4; substi;
++ try inversion H5; substi;
++ inversion H6; substi;
++ 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]

```

```

(*constructor*) to the left of (*the right*) */

-- destruct c; destruct c2;
++ destruct tr2l; destruct tr2r;
try destruct c; try destruct c@;
try redprop_contra;
unfold_balance; unfold make_black;
auto with db;
inversion H1; subst;
inversion H4; subst;
try inversion H5; subst;
inversion H6; subst;
auto with db;

++ destruct tr2l; destruct tr2r;
try destruct c; try destruct c@;
try redprop_contra;
try isblack_contra;
unfold_balance at 1;
auto with db;
inversion H1; subst;
inversion H4; subst;
try inversion H5; subst;
inversion H6; subst;
auto with db;

unfold_balance; unfold make_black;
inversion H6;
constructor; auto with db.

-- auto with db; unfold make_black;
auto with db,
unfold_balance; destruct c;
auto with db; unfold make_black;

```

Read Soft Found 3. and retry

intermediary data structures

```
Inductive RRBBlack {X} :: tree X -> nat -> Prop :=  
| rr_black_node : forall l r v n,  
  RedBlack l n ->  
  RedBlack r n ->  
  RRBBlack (Node Red l v r) n  
| rr_black_black_node : forall l r v n,  
  RedBlack l n ->  
  RedBlack r n ->  
  RRBBlack (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) _ =>  
  | |- RRBBlack _ _ => constructor  
  | |- RedBlack _ _ => constructor  
  | |- _ (match ?c with Red => _ | Black => _ | end) _ => destruct  
  | |- _ (match ?t with Leaf => _ | Node _ _ _ _ => _ | end) _ =>  
  | H : RRBBlack 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 changes

Pain points

- Proof too 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 :: BSTLeaf |- _ => inv H
  | H :: BST (Node _ _ _ _) |- _ => inv H
  | H :: cmp ?y ?x = 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 (redder _) => 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 _ : (redder _) => 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 _ ?y = Gt) ?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 !