FORMAL LAND

FOSDEM - FORMAL VERIFICATION

FEBRUARY 2026

https://formal.land/

ME

Guillaume Claret

FORMAL LAND

MOTIVATION

Testing shows the presence, not the absence of bugs.

Edsger Dijkstra

FORMAL LAND

MOTIVATION

Explosion of Ariane 5 due to a bug

FORMAL LAND

MOTIVATION

Geopolitics

FORMAL LAND

MOTIVATION

Vibe coding

FORMAL LAND

FORMAL VERIFICATION

Mathematical methods to ensure a program is **safe/bug-free** for **all possible inputs**

FORMAL LAND

SURPRISE 1

Bug-free: for a given specification

FORMAL LAND

SURPRISE 2

Mathematical

FORMAL LAND

SURPRISE 3

All possible inputs

FORMAL LAND

ROCQ LANGUAGE

```
(* How list is defined *)
Inductive list (A : Type) : Type :=
  | nil : list A                          (* Empty list *)
  | cons : A → list A → list A.           (* Add element to front *)

(* Notation:
   [] for nil
   x :: xs for cons x xs
   [1; 2; 3] for 1 :: 2 :: 3 :: []
*)
```

FORMAL LAND

ROCQ LANGUAGE

```
(* Length of a list *)
Fixpoint length {A : Type} (l : list A) : nat :=
  match l with
  | [] ⇒ 0
  | _ :: t ⇒ S (length t)
  end.

Compute length [1; 2; 3; 4].  (* = 4 *)
```

FORMAL LAND

ROCQ LANGUAGE

```
(* Append two lists *)
Fixpoint app {A : Type} (l1 l2 : list A) : list A :=
  match l1 with
  | [] ⇒ l2
  | h :: t ⇒ h :: app t l2
  end.

(* Notation: l1 ++ l2 *)
Compute [1; 2] ++ [3; 4].  (* = [1; 2; 3; 4] *)
```

FORMAL LAND

ROCQ LANGUAGE

```
(* Length of append *)
Theorem app_length : forall (A : Type) (l1 l2 : list A),
  length (l1 ++ l2) = length l1 + length l2.
Proof.
  intros A l1 l2.
  induction l1 as [| h t IHt].
  - simpl. reflexivity.
  - simpl. rewrite IHt. reflexivity.
Qed.
```

FORMAL LAND

call_helpers.v

revm > revm_interpreter > instructions > contract > simulate > call_helpers.v

```
Definition resize_memory
  else
    (
      interpreter
    )))).

Lemma resize_memory_eq
    {WIRE : Set} `{Link WIRE}
    {WIRE_types : InterpreterTypes.Types.t} `{InterpreterTypes.Types.AreLinks WIRE_types}
    (run_InterpreterTypes_for_WIRE : InterpreterTypes.Run WIRE WIRE_types)
    (IInterpreterTypes : InterpreterTypes.C WIRE_types)
    (InterpreterTypesEq :
      InterpreterTypes.Eq.t WIRE WIRE_types run_InterpreterTypes_for_WIRE IInterpreterTypes)
    (interpreter : Interpreter.t WIRE WIRE_types)
    (offset len : aliases.U256.t)
    (stack : Stack.t) :
  let ref_interpreter := make_ref 0 in
  let result := resize_memory interpreter offset len in
  {{
    SimulateM.eval_f (
      run_resize_memory run_InterpreterTypes_for_WIRE ref_interpreter offset len
    )
    (interpreter :: stack)%stack 🌲
    (
      Output.Success (fst result),
      (snd result :: stack)%stack
    )
  }}.
Proof.
  apply Run.remove_extra_stack1.
  with_strategy transparent [run_resize_memory] unfold run_resize_memory; cbn.
  unfold resize_memory.
  lu. c. {
    apply Impl_Uint.as_limbs_eq; repeat unshelve econstructor.
  }
  lu. cw Impl_usize.max_eq.
  repeat cp.
  destruct Bool.eqb eqn:?; r. {
    lu. cw InterpreterTypesEq.
    admit.
  }
  lu. repeat cp.
  destruct Bool.eqb eqn:? in ⊢*; r.
  {
    lu. c. {
      apply Impl_Uint.as_limbs_eq; repeat unshelve econstructor.
    }
    lu. cw Impl_usize.max_eq.
    repeat cp.
    destruct Bool.eqb eqn:? in ⊢*; r. {
```

ProofView: call_helpers.v

MAIN 1 SHELVED 2 GIVEN UP 0

```
WIRE : Set
H : Link WIRE
WIRE_types : InterpreterTypes.Types.t
H0 : InterpreterTypes.Types.AreLinks WIRE_types
run_InterpreterTypes_for_WIRE : InterpreterTypes.Run WIRE WIRE_types
IInterpreterTypes : InterpreterTypes.C WIRE_types
InterpreterTypesEq : InterpreterTypes.Eq.t WIRE WIRE_types run_InterpreterTypes_for_WIRE IInterpreterTypes
interpreter : Interpreter.t WIRE WIRE_types
offset, len : aliases.U256.t
stack : Stack.t
```

```
(1/1)
{{SimulateM.eval
    (evaluate
       (lib.Impl_Uint.run_as_limbs {| Integer.value := 256 |} {| Integer.value := 4 |}
          (Ref.cast_to Pointer.Kind.Ref {| Ref.core := Ref.Core.Immediate (Some len) |}))
       .(Run.run_f))
    [interpreter]%stack
🌲 (Output.Success ?output_inter, ?stack_inter)}}
```

FOR WHO?

- Space industry
- Transport
- Defense
- Finance (blockchain)

FORMAL LAND

USE

Limited due to costs

FORMAL LAND

HOPES

- Technical progress
- AI

FORMAL LAND

NEW TREND

- Writing the specification
- Letting the AI do the rest

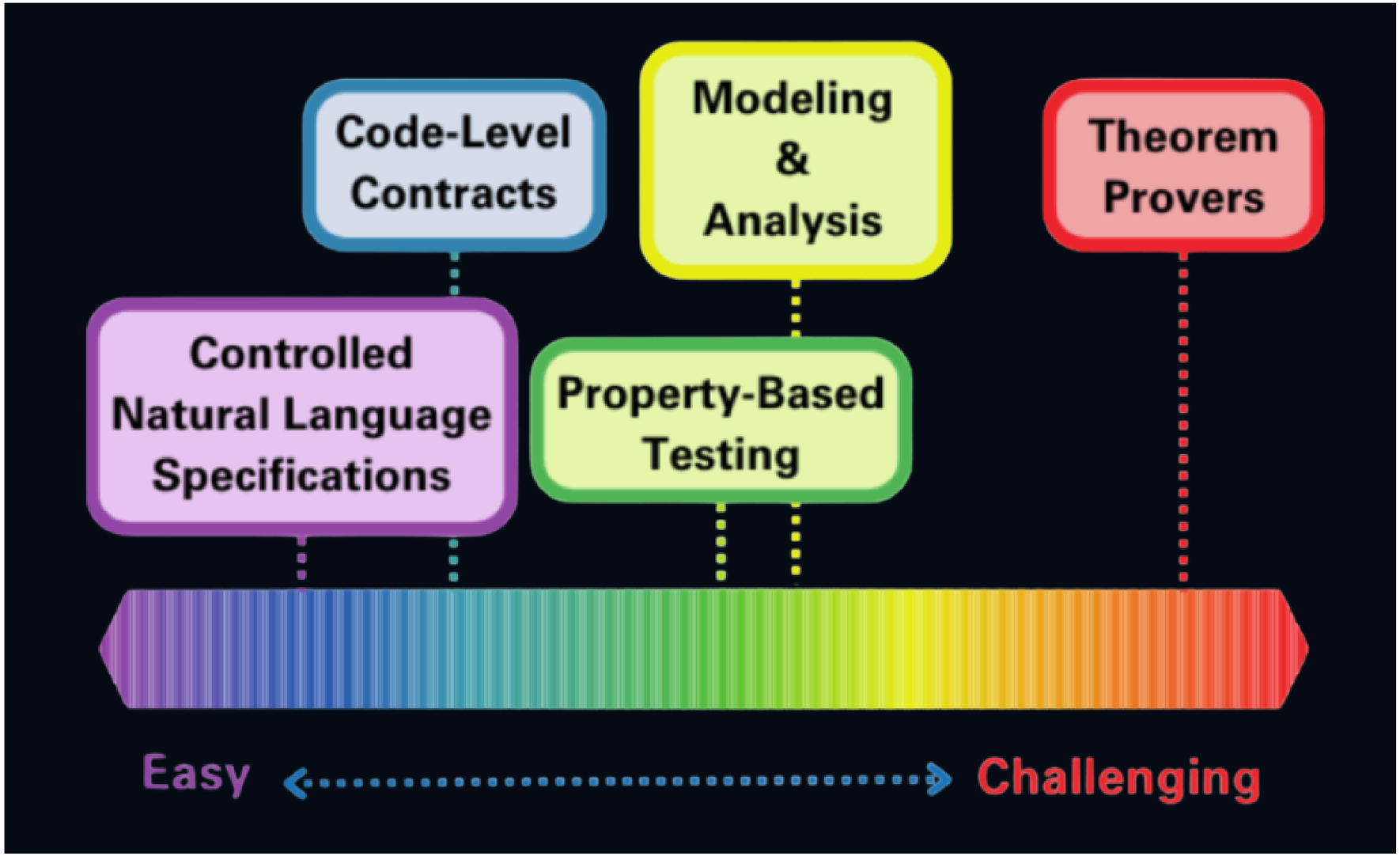
FORMAL LAND

PROPERTIES

- No runtime errors
- No regressions
- Functional correctness
- Business logics

FORMAL LAND

MANY METHODS



MANY PROVERS

- Automated (SMT)
- Interactive

FORMAL LAND

INTERACTIVE PROVERS

- Can express anything
- Can verify anything
- Require human/AI interaction

FORMAL LAND

INTERACTIVE PROVERS

- Rocq (Coq)
- Lean
- Agda
- Isabelle

FORMAL LAND

The Calculus of Constructions

Thierry Coquand and Gérard Huet

INRIA

Résumé

Nous présentons le Calcul des Constructions, un formalisme d'ordre supérieur approprié au développement de preuves constructives dans un style de déduction naturelle. Chaque preuve est une λ-expression, typée avec les propositions de la logique sous-jacente. En effaçant les types on obtient un λ-terme pur, qui représente l'algorithme associé à la preuve. La mise en forme normale de ce λ-terme correspond à l'élimination des coupures.

Nous pensons que la correspondance de Curry-Howard entre les propositions et les types est un guide conceptuel important pour Informatique. Dans le cas des Constructions, nous obtenons un language fonctionnel de très haut niveau, avec une notion de polymorphisme appropriée au développement d'algorithmes modulaires. La notion de type comprend la notion usuelle de type de donnée, mais autorise aussi bien l'expression de spécifications algorithmiques arbitrairement complexes.

Nous développons la théorie de base d'un Calcul des Constructions, et prouvons un résultat de normalisation forte impliquant la terminaison des calculs, et la cohérence de la logique. Finalement, nous suggérons diverses extensions possibles.

Une version préliminaire de ce papier a été présentée au Colloque International sur les Types de Sophia-Antipolis en Juin 1984.

Abstract

We present the Calculus of Constructions, a higher-order formalism for constructive proofs in natural deduction style. Every proof is a λ-expression, typed with propositions of the underlying logic. By removing types we get a pure λ-expression, expressing its associated algorithm. Computing this λ-expression corresponds roughly to cut-elimination.

It is our thesis that the Curry-Howard correspondance between propositions and types is a powerful paradigm for Computer Science. In the case of Constructions, we obtain the notion of a very high-level functional programming language, with complex polymorphism well-suited for modules specification. The notion of type encompasses the usual notion of data type, but allows as well arbitrarily complex algorithmic specifications.

We develop the basic theory of a Calculus of Constructions, and prove a strong normalization theorem showing that all computations terminate. The logical consistency is obtained as a corollary. Finally, we suggest various extensions to stronger calculi.

A preliminary version of this paper was presented in June 1984 at the International Symposium on Semantics of Data Types in Sophia-Antipolis.

ROCQ

- CoC, 1984
- Thierry Coquand
- Mix programs and proofs

ROCQ

- Programming language
- Maths language
- Proofs about code
- Code about proofs

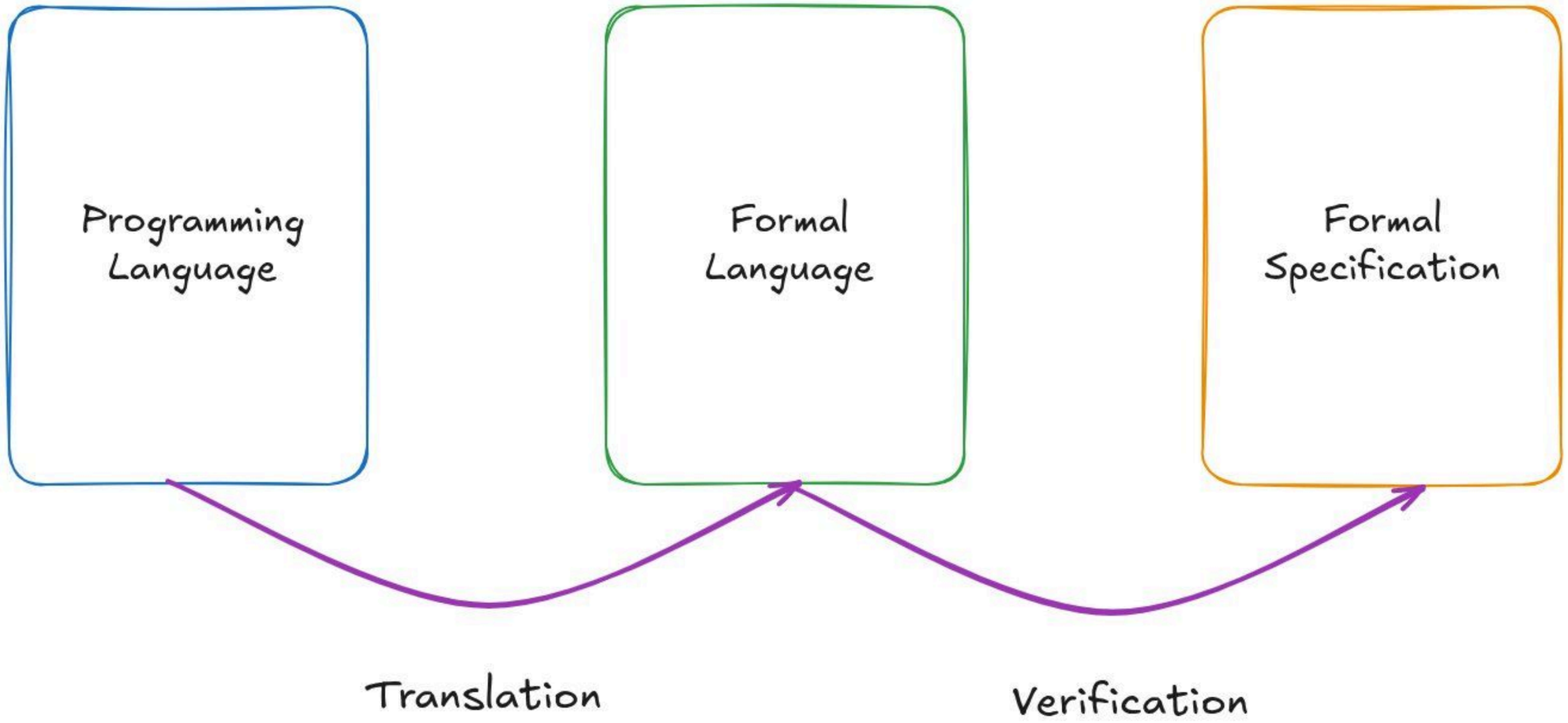
FORMAL LAND

DEPENDENT TYPES

- double a[5];
- double a[2 * n + 1];

FORMAL LAND

IN PRACTICE


Programming Language
Formal Language
Formal Specification
Translation
Verification

LIGHT-WEIGHT VERIFICATION

```rust
// Simple saturating counter in Rust
pub struct Counter {
    pub value: u64,
}

const MAX_VALUE: u64 = 1000;

impl Counter {
    pub fn increment(&mut self, amount: u64) {
        if self.value + amount > MAX_VALUE {
            self.value = MAX_VALUE;
        } else {
            self.value += amount;
        }
    }
}
```

FORMAL LAND

LIGHT-WEIGHT VERIFICATION

```
(* Definition of the Counter state *)
Record counter_state := {
  value : Z
}.

(* Global constant *)
Definition MAX_VALUE : Z := 1000.

(* The functional model of the 'increment' method *)
Definition increment (s : counter_state) (amount : Z) : counter_state :=
  if (s.(value) + amount >? MAX_VALUE) then
    {| value := MAX_VALUE |}
  else
    {| value := s.(value) + amount |}.
```

FORMAL LAND

FULL VERIFICATION

Requires a transpiler

FORMAL LAND

FULL VERIFICATION

- rocq-of-rust
- rocq-of-solidity
- Garden

FORMAL LAND

RUST

```rust
// Simple saturating counter in Rust
pub struct Counter {
    pub value: u64,
}

const MAX_VALUE: u64 = 1000;

impl Counter {
    pub fn increment(&mut self, amount: u64) {
        if self.value + amount > MAX_VALUE {
            self.value = MAX_VALUE;
        } else {
            self.value += amount;
        }
    }
}
```

FORMAL LAND

ROCQ

```
Definition increment (ε : list Value.t) (τ : list Ty.t) (α : list Value.t) : M :=
  match ε, τ, α with
  | [], [], [ self; amount ] ⇒
    ltac:(M.monadic
      (let self :=
        M.alloc (| Ty.apply (Ty.path "&mut") [] [ Ty.path "fosdem::Counter" ], self |) in
      let amount := M.alloc (| Ty.path "u64", amount |) in
      M.match_operator (|
        Ty.tuple [],
        M.alloc (| Ty.tuple [], Value.Tuple [] |),
        [
          fun γ ⇒
            ltac:(M.monadic
              (let γ :=
                M.use
                  (M.alloc (|
                    Ty.path "bool",
                    M.call_closure (|
                      Ty.path "bool",
                      BinOp.gt,
                      [
                        M.call_closure (|
                          Ty.path "u64",
                          BinOp.Wrap.add,
                          [
                            M.read (|
                              M.SubPointer.get_struct_record_field (|
                                M.deref (| M.read (| self |) |),
                                "fosdem::Counter",
                                "value"
                              |)
                            |);
                            M.read (| amount |)
                          ]
                        |);
                        M.read (| get_constant (| "fosdem::MAX_VALUE", Ty.path "u64" |) |)
                      ]
                    |)
                  |)) in
                let _ := is_constant_or_break_match (| M.read (| γ |), Value.Bool true |) in
                M.read (|
                  let~ _ : Ty.tuple [] :=
                    M.write (|
                      M.SubPointer.get_struct_record_field (|
                        M.deref (| M.read (| self |) |),
                        "fosdem::Counter",
                        "value"
                      |),
                      M.read (| get_constant (| "fosdem::MAX_VALUE", Ty.path "u64" |) |)
                    |) in
                  M.alloc (| Ty.tuple [], Value.Tuple [] |)
                |)));
          fun γ ⇒
            ltac:(M.monadic
              (M.read (|
                let~ _ : Ty.tuple [] :=
                  let β :=
                    M.SubPointer.get_struct_record_field (|
                      M.deref (| M.read (| self |) |),
                      "fosdem::Counter",
                      "value"
                    |) in
                  M.write (|
                    β,
                    M.call_closure (|
                      Ty.path "u64",
                      BinOp.Wrap.add,
                      [ M.read (| β |); M.read (| amount |) ]
                    |)
                  |) in
                M.alloc (| Ty.tuple [], Value.Tuple [] |)
              |)))
        ]
      |)))
  | _, _, _ ⇒ M.impossible "wrong number of arguments"
  end.
```

FORMAL LAND

ROCQ

```
Definition increment (counter : Counter.t) (amount : u64) : Counter.t :=
  if counter.(Counter.value) +i amount >i MAX_VALUE then
    counter ◁ Counter.value := MAX_VALUE ▷
  else
    counter ◁ Counter.value := counter.(Counter.value) +i amount ▷.
```

FORMAL LAND

ROCQ

```
Definition CounterSmallerThanMAX_VALUE (counter : Counter.t) : Prop :=
  i[counter.(Counter.value)] ⩽ i[MAX_VALUE].

Lemma counter_stays_smaller_than_MAX_VALUE (counter : Counter.t) (amount : u64) :
  CounterSmallerThanMAX_VALUE counter →
  CounterSmallerThanMAX_VALUE (increment counter amount).
Proof.
  destruct counter as [value].
  destruct value as [value], amount as [amount].
  unfold CounterSmallerThanMAX_VALUE, increment, ">i"; cbn.
  intros.
  destruct (_ >? _) eqn:?; cbn; lia.
Qed.
```

LAST PROJECTS

- Zero-knowledge circuits
- Ethereum VM in Rust

FORMAL LAND

LINKS

- https://github.com/formal-land
- https://formal.land/blog

FORMAL LAND

2026?

- Massive verification of OSS?
- FV as a standard for enterprise projects?

FORMAL LAND

THANKS

FORMAL LAND