A Clean and Lean Circom Formalization

Philosophy for the Real World of a Computer Scientist

Christiano Braga

Universidade Federal Fluminense & ZKForall

Typed Floripa

12/07/2025

Who am I?

- Associate Professor at Universidade Federal Fluminense
- Researcher at ZKForall
- Background: formal semantics of programming languages and formal verification
- Recent years: zero-knowledge proofs and fully homomorphic encryption, really cool stuff!

Testing vs. proving I

- "Testing is asking questions; proof is the answer."
  "We test the possibilities, you get the proof."
  "Test the theory, prove the results."
  "From 'What if?' to 'It is.'"
  "Testing builds evidence, proof builds certainty."
  "Test the limits, prove the potential."
- Property-based testing is pretty cool but is not a proof!

Testing vs. proving II

- To prove we must but how to do it? **Logically, obviously!**
- Many many approaches: first order or higher-order
- In this talk: Type Theory

Type Theory i

- L. E. J. Brower: Intuitionistic logic
  - Rejects the law of excluded middle (LEM): $A \vee \neg A$, but with contradiction $(A \to B) \to ((A \to \neg B) \to \neg A)$ and *ex falso sequitur quodlibet* $\neg A \to (A \to B)$
  - Brouwer observed that LEM was abstracted from finite situations, then extended without justification to statements about infinite collections
  - Constructivism!
- Per Martin-Löf: Intuitionistic type theory
- Haskell Curry and William A. Howard: Curry-Howard isomorphism
  - The Curry–Howard correspondence is the observation that there is an isomorphism between **proof systems** and **models of computation**.

Type Theory ii

- Thierry Coquand and Gérard Huet: Calculus of Constructions
- Frank Pfenning and Christine Paulin-Mohring: Inductively Defined Types in the Calculus of Constructions
- **Lean** (and, most notably, Rocq, formerly known as the Coq Proof Assistant) are based on the Calculus of Inductive Constructions

Lean as a Functional Programming language I

- Key characteristics
  - Lean is a **pure, strict functional language with dependent types**
- Types and Polymorphism
  - Every expression has a **type**
  - Lean supports **parametric polymorphism**

Lean as a Functional Programming language II

- Functions are first-class
- Algebraic datatypes: structures (product types), inductives (sum types, or tagged-union), pattern matching

Lean as a Functional Programming language III

- Safe & terminating recursion
  - Recursive definitions must be **structurally recursive** or come with a proof of termination
- Effects via Monads & type-level effects
  - Effects (I/O, state, errors) are encoded using **monads** and do-notation

Simple Examples i

```
-- Polymorphic function: length of a list
def length {α} (xs : List α) : Nat :=
    match xs with
    | []      => 0
    | _ :: t1 => 1 + length t1

-- Algebraic datatype + pattern matching
inductive MyOption (α : Type) where
  | none : MyOption α
  | some : α → MyOption α

def getOrDefault {α : Type} (opt : Option α) (default : α) : α :=
  match opt with
  | Option.none   => default
  | Option.some x => x

-- Using monad (IO) for effectful code
def main : IO Unit := do
  IO.println s!"Hello, world!" --
```

Lean Tactics

- Tactics are commands used inside a tactic block to incrementally transform goals until the proof is finished

Some Lean Tactics I

Tactic	Purpose	Example
`simp`	Simplifies goal and hypotheses using the simplifier	`simp [only add_assoc]`
`rw`	Rewrites using equalities as rewrite rules	`rw [← add_assoc] at h`

`cases`	Case analysis on inductive values	`cases n`
		`case zero => ...`
		`case succ n' => ...`

Some Lean Tactics II

Tactic	Purpose	Example
`aesop`	Automated reasoning (search + rewriting + rules)	`by aesop`
`grind`	Performs normalization, congruence closure, rewriting, linear arithmetic	`by grind`
`omega`	Solves linear integer/natural Presburger arithmetic	`by omega`

Why do we need proofs to be **formal**?

Why not stick to “pen and paper”?

Natural Deduction Proof for the existence of a prime factor I

Definition of Prime predicate

$\forall n, Prime(n) = 1 < n \wedge \forall k, 1 < k \rightarrow k < n \rightarrow \neg k \mid n$

Theorem. For every natural number (n), if $(1 < n)$, then there exists a prime number (k) such that $(k \mid n)$.

Formally:

$$\forall n, (1 < n \rightarrow \exists k, (\mathsf{Prime}(k) \wedge k \mid n)).$$

Notation $k \mid n$ *means that* $\exists c$ *such that* $n = c \cdot k$

Natural Deduction Proof for the existence of a prime factor II

Proof

Let (n) be an arbitrary natural number and assume

1. $(1 < n)$. (assumption)

We perform a **case distinction** on whether (n) is prime.

Natural Deduction Proof for the existence of a prime factor III

Case 1. (n) **is prime.**

Assume

2. $(\text{Prime}(n))$. (case assumption)

From (2), by **conjunction introduction**, we know

3. $(\text{Prime}(n) \wedge n \mid n)$,

because every number divides itself.

Then by **existential introduction** on (3), we conclude:

4. $(\exists k, (\text{Prime}(k) \wedge k \mid n))$.

This completes Case 1.

Natural Deduction Proof for the existence of a prime factor IV

Case 2. (n) **is not prime.**

Assume

5. $(\neg\text{Prime}(n))$. (case assumption)

Since (n) is not prime and $(1 < n)$, by the definition of non-prime, we obtain:

6. $(\exists k, (1 < k \wedge k < n \wedge k \mid n))$. ($\exists$-introduction from the definition of composite)

Let

7. (k) be such that
   - $(1 < k)$,
   - $(k < n)$,
   - $(k \mid n)$. (from 6 by $\exists$-elimination)

Natural Deduction Proof for the existence of a prime factor V

From $(1 < k)$, by **induction on smaller numbers** (or, in classical math: by applying the lemma recursively to a strictly smaller number), we may apply the theorem to (k). Since $(k < n)$ and $(1 < k)$, we get:

8. $(\exists p, (\text{Prime}(p) \land p \mid k))$. (by IH / recursive application)

Choose such a (p). So we have:

9. $(\text{Prime}(p))$ and $(p \mid k)$. (from 8 by $\exists$-elimination)

From $(p \mid k)$ and $(k \mid n)$, by the **transitivity of divisibility**, we obtain:

10. $(p \mid n)$. (divisibility transitivity)

Natural Deduction Proof for the existence of a prime factor VI

Now by **existential introduction** applied to prime (p):

11. $(\exists p, (\text{Prime}(p) \wedge p \mid n))$.

This completes Case 2.

Natural Deduction Proof for the existence of a prime factor VII

Conclusion
Since from the assumption $(1 < n)$, both cases—either (n) is prime or not—lead to the conclusion

$$\exists k, (\mathsf{Prime}(k) \land k \mid n),$$

we may discharge the case distinction and conclude by **implication introduction**:

$$1 < n \to \exists k, (\mathsf{Prime}(k) \land k \mid n).$$

Finally, by **universal introduction** (since (n) was arbitrary):

$$\forall n, (1 < n \to \exists k, (\mathsf{Prime}(k) \land k \mid n)). \quad \square$$

Existence of Prime factor in Lean

```
/-- A prime is a number larger than 1 with no trivial divisors -/
def IsPrime (n : Nat) := 1 < n ∧ ∀ k, 1 < k → k < n → ¬ k ∣ n

/-- Every number larger than 1 has a prime factor -/
theorem exists_prime_factor :
    ∀ n, 1 < n → ∃ k, IsPrime k ∧ k ∣ n := by
  intro n h1
  -- Either `n` is prime...
  by_cases hprime : IsPrime n
  · grind [Nat.dvd_refl]
  -- ... or it has a non-trivial divisor with a prime factor
  · obtain ⟨k, _⟩ : ∃ k, 1 < k ∧ k < n ∧ k ∣ n := by
      simp_all [IsPrime]
    obtain ⟨p, _, _⟩ := exists_prime_factor k (by grind)
    grind [Nat.dvd_trans]
```

Now we know how to formally prove in Type Theory

What about ZKP circuits?

Zero-knowledge proofs i

- Goldwasser, Micali and Rackoff's zero-knowledge proof (ZKP) is a cryptographic concept
- A prover convinces a verifier that a statement is true without revealing why it is true
- The prover shows knowledge of a secret witness (e.g., a password, preimage of a hash, a transaction's private data) that satisfies some publicly known constraints
- But the verifier learns nothing beyond the fact that the constraints are indeed satisfiable
- ZKPs must satisfy completeness (honest proofs pass), soundness (cheaters fail), and zero-knowledge (the proof reveals nothing)

ZK Circuits

- To prove a statement in zero-knowledge, a (class of) computation(s) is formalized as **arithmetic circuits** over a finite field composed of small, verifiable arithmetic steps
  - A proof is about one such computation
- Its components are:
  - Inputs: public inputs + private witness
  - Gates: simple field operations (addition, multiplication)
  - Constraints: equations that must hold for the witness to be valid
  - Output: a boolean condition (valid/invalid)

Circom

- Circom is a domain-specific language (DSL) for describing (field) arithmetic circuits used in zk-SNARKs

Circom Core Ideas I

- Everything is a circuit:
  - Circom programs are compositions of components, each with inputs, outputs, and internal signals
- Signals = wires:
  - Signals carry field elements (in the prime field of the proof system)
  - Assigning signal = expr does not compute a value — it declares a constraint

Circom Core Ideas II

- Constraints are equations defined with ===

```
a * b === c;
a + b === 7;
```

- Components are reusable:

```
component add = Add();
add.a <== x;
add.b <== y;
out <== add.out;
```

Circom Core Ideas III

- Assignment operators:
  - `===` $\rightarrow$ constraint
  - `<==` $\rightarrow$ wiring/connection of signals
  - `<--` $\rightarrow$ witness assignment (for template-internal computation)
- Templates = parametric circuits:

```
template AddN(n) {
    signal input in[n];
    signal output out;
    ...
}
```

Circom Core Ideas IV

- Compilation flow:
  - Circom $\rightarrow$ Rank 1 Constraint System (R1CS)
  - Witness generation (input assignment)
  - Zero-Knowledge Succinct Non-interactive Argument of Knowledge (zk-SNARK) proof generation

Circom Example

- Circomlib/Comparators.lean#L15

```
template IsZero() {
    signal input in;
    signal output out;

    signal inv;

    inv <-- in != 0 ? 1 / in : 0;

    out <== -in * inv + 1;
    in * out === 0;
}
```

Clean

- Clean is a high-level domain-specific language (DSL), implemented in Lean, designed to describe arithmetic circuits for zero-knowledge proofs in a modular way
- Essentially circuits are instances of a monad of constraints
- Clean defines its own tactics, e.g. `circuit_proof_start`

Clean Circuit I

- Circuit
- A circuit is a *monad*, an abstraction for programming with effects in functional programming, borrowed from Category Theory by Eugenio Moggi
- Concretely, a `Circuit` is a function `(offset : ℕ) → α × List (Operation F)` for some return type α

```
def Circuit (F : Type) [Field F] (α : Type) := ℕ → α × List (Operation F)
```

Clean Circuit II

- The monad is a mix of:
  - a writer monad that accumulates the list of operations
  - a state monad that keeps track of the offset
  - where the next offset is computed from the operations added in the previous step

Correctness of a Clean Circuit I

- `FormalCircuit`
- It requires:
  - a spec, which is a relationship between inputs and outputs
  - assumptions, which are the conditions that must hold for the circuit to make sense
  - a proof of *soundness*: assumptions $\wedge$ constraints $\rightarrow$ spec, for any witness
  - a proof of *completeness*: assumptions $\rightarrow$ constraints, using some existing witness

Correctness of a Clean Circuit II

- When viewed as a black box, the circuit acts similarly to a function
- The assumptions act as preconditions, and the spec acts as the postcondition

```
structure FormalCircuit (F : Type) [Field F]
                        (Input Output : TypeMap)
                        [ProvableType Input] [ProvableType Output]
    extends elaborated : ElaboratedCircuit F Input Output where
  Assumptions (_ : Input F) : Prop := True
  Spec : Input F → Output F → Prop
  soundness : Soundness F elaborated Assumptions Spec
  completeness : Completeness F elaborated Assumptions
```

Correctness of `isZero` I

- Main function

Circom	Clean

```
template IsZero() {
   signal input in;
   signal output out;
   signal inv;
   inv <-- in!=0 ? 1/in : 0;
}
```

```
def main (input : Expression (F p)) := do
    let inv ← witness fun env =>
        let x := input.eval env
        if x != 0 then x^-1 else 0
    let out <== -input * inv + 1
    out <== -in*inv +1;
    in*out === 0;
    return out
```

Correctness of isZero II

- Circuit

```
namespace IsZero

def main(...) := ...

def circuit : FormalCircuit (F p) field field where
  main
  localLength _ := 2

  Assumptions _ := True

  Spec input output :=
    output = (if input = 0 then 1 else 0)

  soundness := ...
  completeness := ...

end IsZero
```

Correctness of isZero III

- Soundness

```
soundness := by
  circuit_proof_start
  simp only [id_eq, h_holds]
  split_ifs with h_ifs
  . simp only [h_ifs, zero_mul, neg_zero, zero_add]
  . rw [neg_add_eq_zero]
    have h1 := h_holds.left
    have h2 := h_holds.right
    rw [h1] at h2
    simp only [id_eq, mul_eq_zero] at h2
    cases h2
    case neg.inl hl => contradiction
    case neg.inr hr =>
      rw [neg_add_eq_zero] at hr
      exact hr
```

Correctness of `isZero` IV

- Completeness

```
completeness := by
  circuit_proof_start
  cases h_env with
  | intro left right =>
    simp only [left, ne_eq, id_eq, ite_not, mul_ite, mul_zero] at right
    simp only [id_eq, right, left, ne_eq, ite_not,
                                mul_ite, mul_zero, mul_eq_zero, true_and]
    split_ifs <;> aesop
```

Coda I

- The objective of this talk was to talk about Type Theory, the Lean system and ZK Security's Clean approach to ZKP circuit verification.

Coda II

- There are other approaches to verify circuits in other ZKP DSL (and proof systems).
- Notably, Nethermind's:
  - Plonky 3: CertiPlonk
  - Halo 2: Halva

Coda III

- Also worth mentioning is the combination of AI and (Lean) theorem proving by:
  - Google Deepmind's AlphaProof
  - Harmony's Aristotle

Coda IV

- Another very interesting technique to enforce privacy is *fully homomorphic encryption*, where the effect of operations applied to encrypted data is preserved in the decrypted data.
- How do we verify such constructions?

I guess we have a lot to talk about! =)

Thank you!

A Clean and Lean Circom Formalization

Philosophy for the Real World of a Computer Scientist

Christiano Braga

Universidade Federal Fluminense & ZKForall

Typed Floripa

12/07/2025