Intro to Turn-Lang's formal library - Turn-Formal

A path to formalize critical subjects (in Rust)

Turner, Creator of turn-lang.com

2025-04-19

Abstract

Turn-Formal is a new approach to formal mathematics and verification, built in Rust. This paper introduces the Turn-Formal library, its architecture, and its advantages over existing systems. We discuss the motivation behind high-level formal systems, demonstrate how Turn-Formal makes verification accessible to developers, and outline our roadmap for future development. Turn-Formal aims to bridge the gap between rigorous mathematical proof and practical software development, providing an expressive yet powerful framework for formal verification.

Contents

1 Introduction to Formal Systems

Formal systems provide a foundation for rigorous mathematical reasoning with machine-checkable proofs.

Key characteristics of formal systems:

- Precise, unambiguous language and syntax
- Well-defined rules of inference
- Mechanically verifiable proofs
- Foundation for automated theorem proving

Why formalization matters:

- Eliminates ambiguity in mathematical proofs
- Enables machine verification of correctness
- Facilitates the development of verified software
- Provides a basis for advanced AI reasoning systems
- Bridges the gap between mathematical theory and practical applications

Applications of formalization span numerous domains:

- Verifying correctness of cryptographic protocols
- Ensuring safety-critical systems meet specifications
- Developing certified compilers and software
- Advancing mathematical knowledge through verified proofs
- Creating foundations for AI reasoning and decision-making

2 High-Level Formal Mathematics

Traditional formal systems often operate at a low level of abstraction, making them challenging to use for everyday mathematical practice.

High-level formal mathematics aims to:

- Match the intuition and workflow of human mathematicians
- Abstract away mechanical details while maintaining rigor
- Provide a natural language-like experience for formal proof development
- Enable mathematicians to work at their level of conceptual understanding
- Bridge the gap between informal mathematical practice and formal verification

The abstraction gap:

- Traditional formal systems: Detailed, granular steps that are machine-friendly but human-hostile
- Informal mathematics: Intuitive leaps that are human-friendly but machine-hostile
- High-level formal systems: The ideal middle ground that serves both humans and machines

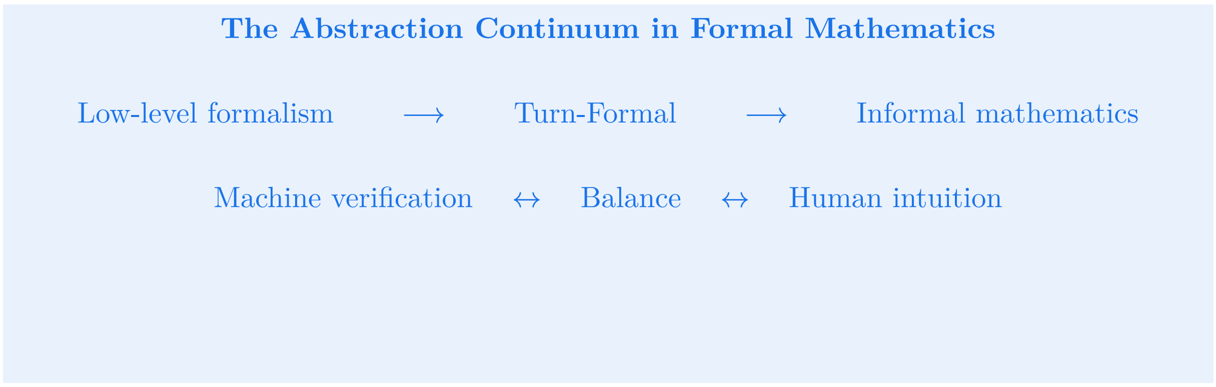
The Abstraction Continuum in Formal Mathematics
Low-level formalism ⟶ Turn-Formal ⟶ Informal mathematics
Machine verification ↔ Balance ↔ Human intuition

Figure 1: The position of Turn-Formal in the spectrum of mathematical formalization approaches

3 Turn-Formal vs. Existing Systems

Turn-Formal, implemented in Rust, offers significant advantages over existing systems like Lean4:

Key advantages:

- **Performance**: Rust's speed and memory safety combine rigor with efficiency
- **Accessibility**: Domain-specific language makes formal mathematics more approachable
- **Integration**: Seamless interoperability with the broader Rust ecosystem
- **Modularity**: Flexible architecture for various mathematical domains
- **Expressiveness**: Rich syntax for intuitive theorem statements and proofs

Comparison with Lean4:

- More intuitive syntax for common mathematical constructs
- Stronger performance characteristics for large-scale formalization
- Better integration with production programming environments
- Focus on developer experience and practical applications
- Built-in support for domain-specific mathematical theories

The table below provides a feature comparison between Turn-Formal and other popular formal systems:

Feature	Turn-Formal	Lean4	Coq
Implementation language	Rust	Lean	OCaml
Memory safety	Native	Runtime	Runtime
Metaprogramming	Rust macros	Meta-Lean	Ltac
Proof style	Tactic & declarative	Tactic & term	Tactic-focused
Learning curve	Moderate	Steep	Steep
Integration with standard tools	Strong	Limited	Limited
Performance	High	Moderate	Moderate

Table 1: Comparison of Turn-Formal with other formal verification systems

4 Developer-Friendly Formal Verification

Turn-Formal is designed with developers in mind, making formal verification accessible to software engineers.

Developer-friendly features:

- Familiar Rust syntax and semantics
- Strong type system that catches errors early
- Flexible tactics system that can be extended for specific domains
- Clear, chainable API for proof construction
- Comprehensive documentation and examples

Core components:

- **ProofState**: Represents the current state in a formal proof
- **Tactics**: Operations that transform proof states (like introduction, substitution)
- **TheoremBuilder**: Constructs formal theorems with structured proofs
- **ProofBranch**: Manages different paths in a proof exploration
- **MathRelation**: Represents mathematical relationships

Turn-Formal's API is designed to be intuitive and chainable:

Example: Creating and building a proof

```rust
let state = ProofState::new();
let branch1 = state
    .tactics_intro_expr("a", MathExpression::Var(Identifier::E(1)), 0)
    .tactics_intro_expr("b", MathExpression::Var(Identifier::E(2)), 1);

// Add some proof steps
let p1 = branch1.tactics_intro_expr("a", create_var("a"), 1);
let p2 = p1.tactics_intro_expr("b", create_var("b"), 2);
```

```
// Mark as complete
let p3 = p2.should_complete();
```

The architecture of Turn-Formal follows a layered approach:

User-Facing API ProofBranch, TheoremBuilder, Tactics
Proof Structure ProofForest, ProofNode, CaseAnalysis
Mathematical Foundation MathRelation, MathExpression, Identifiers
Rust Core Memory Safety, Performance, Ecosystem

Table 2: The layered architecture of Turn-Formal

5 Creating Theorems with Turn-Formal

Turn-Formal makes it easy to express and prove theorems in various mathematical domains. Below is an example of proving a theorem in group theory:

Example: Proving a theorem about group inverses

```
// Prove that in a group, inverses are unique
pub fn prove_inverse_uniqueness() -> Theorem {
    // Create a group structure for our proof
    let group = create_abstract_group();

    // Create element variables
    let g_var = create_element_variable(&group, "g", 1);
    let h1_var = create_element_variable(&group, "h1", 2);
    let h2_var = create_element_variable(&group, "h2", 3);
    let e_var = GroupExpression::identity(group.clone());

    // Create relations for our proof
    let relation1 = group_operation_equals(&group, &g_var, &h1_var, &e_var);
    let relation2 = group_operation_equals(&group, &g_var, &h2_var, &e_var);

    // Create the theorem statement: if g*h1 = e and g*h2 = e, then h1 = h2
    let theorem_statement = MathRelation::Implies(
        Box::new(MathRelation::And(vec![
            relation1.clone(),
            relation2.clone(),
        ])),
        Box::new(MathRelation::equal(
            h1_var.to_math_expression(),
            h2_var.to_math_expression(),
        )),
    );

    // Build the proof
    let builder = TheoremBuilder::new("Group Inverse Uniqueness",
theorem_statement, vec![]);

    // Initial branch
    let p0 = builder.initial_branch();

    // ... proof steps ...

    // Build the theorem
    builder.build()
}
```

This example demonstrates how Turn-Formal enables developers to express mathematical concepts directly in Rust, with a clear and readable syntax that closely mirrors mathematical notation while maintaining the full power of formal verification.

6 Future Roadmap

The roadmap for Turn-Formal focuses on expanding capabilities and accessibility:

Short-term goals:

- Complete foundational mathematics library
- Enhance tactic system with machine learning suggestions
- Improve proof visualization and exploration tools
- Develop integrations with common IDEs

Medium-term goals:

- Build domain-specific libraries for cryptography, distributed systems
- Create automated proof search capabilities
- Develop translation layers for interoperability with other proof assistants
- Establish a package ecosystem for community contributions

Long-term vision:

- Formal verification as a standard part of software development
- Bridge between theorem provers and mainstream programming
- Accessible formal methods for non-specialists
- Unified platform for mathematical formalization across disciplines

Turn-Formal Development Timeline

Phase 1: Foundation	**Phase 2:** Expansion	**Phase 3:** Integration
Core tactics	Domain libraries	Industry adoption
Basic proof system	ML assistance	Education resources
Foundational theorems	Interactive tools	Ecosystem growth

Table 3: Development phases for the Turn-Formal project

As we continue to develop Turn-Formal, we invite the community to join us in building a more rigorous and reliable software ecosystem, where formal verification becomes a standard part of the development process rather than a specialized niche.

References

Bibliography