Math-IR: A Compiler-Inspired Approach to Mathematical Formalization

> LaTeX → S-Expression IR → Lean 4

Overview

Math-IR is a prototype system that applies compiler design principles (inspired by MLIR/IREE) to the problem of formalizing mathematical statements. It provides a **homoiconic intermediate representation** for mathematical expressions that can be:

1. **Extracted** from LaTeX using LLMs
2. **Validated** structurally and semantically
3. **Transformed** through progressive lowering passes
4. **Emitted** to formal proof assistants (Lean 4)
5. **Verified** against established libraries (Mathlib)

Architecture

```
┌─────────────────────────────────────────────────────────────┐
│                    MATH-IR ARCHITECTURE                     │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  ┌─────────────┐                                            │
│  │   LaTeX     │  "x^2 \geq 0"                              │
│  │  (source)   │                                            │
│  └──────┬──────┘                                            │
│         │                                                   │
│         │ LLM Extraction (Claude)                           │
│         ▼                                                   │
│  ┌─────────────┐                                            │
│  │    IR       │ {:node :app :op :ge                        │
│  │   (EDN)     │   :args [{:node :app :op :pow ...} ...]}   │
│  └──────┬──────┘                                            │
│         │                                                   │
│         │ Validation + Transformation                       │
│         ▼                                                   │
│  ┌─────────────┐                                            │
│  │   Lean 4    │  "x ^ 2 ≥ (0 : ℝ)"                         │
│  │  (target)   │                                            │
│  └──────┬──────┘                                            │
│         │                                                   │
│         │ Type Checking (Mathlib)                           │
│         ▼                                                   │
│  ┌─────────────┐                                            │
│  │  Verified   │  ✓ or ✗ + error feedback                   │
│  │   Result    │                                            │
│  └─────────────┘                                            │
│                                                             │
└─────────────────────────────────────────────────────────────┘
```

IR Schema

The IR uses a homoiconic S-expression format (EDN in Clojure, nested dicts in Python):

Node Types

Node	Description	Example
`:var`	Variable reference	`{:node :var :name "x" :type :Real}`
`:lit`	Literal value	`{:node :lit :value 2 :type :Nat}`
`:app`	Binary operator	`{:node :app :op :add :args [left right]}`
`:unary`	Unary operator	`{:node :unary :op :abs :arg expr}`
`:fn-app`	Function application	`{:node :fn-app :fn "f" :args [x]}`
`:quant`	Quantifier	`{:node :quant :kind :forall :bindings [...] :body expr}`

Operators

Arithmetic: `:add`, `:sub`, `:mul`, `:div`, `:pow`, `:neg`

Unary: `:abs`, `:sqrt`, `:exp`, `:log`

Relations: `:eq`, `:ne`, `:lt`, `:le`, `:gt`, `:ge`

Logical: `:and`, `:or`, `:implies`, `:iff`, `:not`

Quantifiers: `:forall`, `:exists`

Types

- `:Nat` - Natural numbers
- `:Int` - Integers
- `:Real` - Real numbers
- `:Prop` - Propositions

Design Decisions

These conventions resolve LaTeX ambiguities:

Ambiguity	Resolution
Implicit multiplication (`2ab`)	Always expand to explicit `:mul`
Operator arity	Binary only; nest for n-ary
Chained comparisons (`a < b < c`)	Desugar to conjunction
Literal types	Exponents → `:Nat`, otherwise → `:Real`
Binding constraints (`ε > 0`)	Separate `:constraint` field

Running the Demo

Python

```bash
cd math-ir
python3 run.py
```

Clojure (requires Babashka)

```bash
cd math-ir
bb run.clj
```

Project Structure

```
math-ir/
├── src/
│   ├── schema.cljc      # Core IR schema and constructors
│   ├── emit_lean.cljc   # Lean 4 code generator
│   ├── extract.cljc     # LLM-based extraction
│   ├── pipeline.cljc    # Main pipeline orchestration
│   └── corpus.cljc      # Test corpus (20 expressions)
├── output/
│   └── generated.lean   # Sample Lean output
├── run.py               # Python demo
├── run.clj              # Clojure demo
└── README.md
```

Test Corpus

The corpus includes 20 expressions across 4 difficulty tiers:

Tier 1: Simple

- $x^2 \geq 0$
- $a + b = b + a$
- $|x| \geq 0$
- $\sqrt{4} = 2$
- $0 < 1$

Tier 2: Basic Structure

- $(a/b) \cdot (c/d) = ac/bd$
- $(a + b)^2 = a^2 + 2ab + b^2$
- $|xy| = |x||y|$
- $\sqrt{(x^2)} = |x|$
- $a^m \cdot a^n = a^{m+n}$

Tier 3: Nested/Complex

- $|x + y| \leq |x| + |y|$
- $a/b + c/d = (ad + bc)/bd$
- $\sqrt{(a/b)} = \sqrt{a}/\sqrt{b}$
- $(a-b)(a+b) = a^2 - b^2$
- $|a/b| = |a|/|b|$

Tier 4: Quantified

- $\forall x \in \mathbb{R}, x^2 \geq 0$
- $\forall a,b \in \mathbb{R}, |a+b| \leq |a| + |b|$
- $\exists x \in \mathbb{R} : x^2 = 2$
- $\forall \varepsilon > 0, \exists \delta > 0 : |x| < \delta \rightarrow |f(x)| < \varepsilon$
- $\forall x,y \in \mathbb{R}, x < y \rightarrow \exists z : x < z < y$

Roadmap

Phase 1: Foundation (Current)

- [x] Define IR schema
- [x] Implement Lean emitter
- [x] Create test corpus
- [x] Build validation pipeline
- [x] Python + Clojure implementations

Phase 2: LLM Integration

- [ ] Structured output extraction prompts
- [ ] Chain-of-thought disambiguation
- [ ] Error-driven refinement loop
- [ ] Confidence calibration

Phase 3: Verification Loop

- [ ] Lean type-checker integration
- [ ] Error feedback parsing
- [ ] Automatic retry with corrections
- [ ] Mathlib coverage mapping

Phase 4: Grammar Learning

- [ ] Rule induction from successful parses
- [ ] Grammar evolution algorithm
- [ ] Profinite approximation via Mathlib strata
- [ ] Active learning for edge cases

Theoretical Foundation

This project embodies several key insights:

MLIR-Inspired Dialect Hierarchy

Just as MLIR uses progressive lowering through dialects, Math-IR envisions:

```
theorem.source (LaTeX-close)
       ↓
math.semantic (typed, unambiguous)
       ↓
type.dependent (fully elaborated)
       ↓
lean.surface (target syntax)
```

Profinite Grammar Space

The space of valid IR grammars forms a profinite structure:

- Each Mathlib theorem defines a verification constraint
- Grammar refinement = projective limit as theorems accumulate
- The "optimal grammar" is what survives all verification tests

Eigenobject Perspective

The optimal grammar is the **eigenobject** under the verification functor—it's what persists when we require consistent translation across all notational variants of the same mathematical content.

References

- **MLIR**: mlir.llvm.org
- **IREE**: iree.dev
- **Mathlib**: leanprover-community.github.io
- **Lean 4**: lean-lang.org

License

MIT

Contributing

This is a research prototype. Contributions welcome for:

- Expanding the test corpus
- Improving the Lean emitter
- Adding support for more mathematical constructs
- Implementing the LLM extraction pipeline