Gödel Mirror

A Paraconsistent Calculus that Metabolizes Contradictions Mechanized in Lean 4

Jhet Chan

Independent Researcher, Malaysia

ITP 2025 Lean Workshop

2 October 2025

Reykjavík, Iceland

Motivation

Contradictions are Everywhere

- **Large Language Models (LLMs)** often contradict themselves.
- In classical logic, a contradiction ($P \land \neg P$) leads to **explosion**, anything can be proven.
- **Gödel's Incompleteness Theorems** showed that paradox is inevitable in any system powerful enough for self-reference.

Our Question:

Instead of letting contradictions explode, can we **metabolize** them as part of the computation?

The Gödel Mirror Mechanism

From Explosion to a Controlled Cycle

The Gödel Mirror turns a contradictions into a stable, structured object through a three-step cycle.

- **Paradox:** A self-referential term is identified.
- **cap(...):** The paradox is encapsulated, preventing explosion.
- **enter(...):** The capsule re-enters the system in a controlled way.
- **node(...):** It stabilizes into a structured "node", a value.

The key idea:
Contradiction becomes a stable object in computation (read: if $P \land \neg P \rightarrow$ recursion)

Encoding in Lean

The entire system is captured in a simple inductive data type.

```
inductive MirrorSystem where
  | base
  | node    : MirrorSystem → MirrorSystem
  | self_ref -- Represents a paradoxical term
  | cap     : MirrorSystem → MirrorSystem
  | enter   : MirrorSystem → MirrorSystem
  | named   : String → MirrorSystem → MirrorSystem
```

Live Demo: The Liar Paradox (Demo.lean)

Let's define the Liar paradox: "This statement is false", represented by a named self-reference.

```
def liar := MirrorSystem.named "Liar" MirrorSystem.self_ref

-- Run it for 3 steps
#eval run liar 3
```

Demo repo (branch ‘itp’):

https://github.com/jhetchan/godel-mirror/tree/itp

Demo 1: Liar Paradox (`named "Liar" self_ref`)

```
Step 0: 🏷️ 'Liar'                                    ← Start: paradoxical term
Step 1: 🔥 cap(🏷️ 'Liar')                            ← Encapsulated (paradox wrapped)
Step 2: 🌱 enter(🔥 cap(🏷️ 'Liar'))                  ← Reentering
Step 3: 🔁 node(🌱 enter(🔥 cap(🏷️ 'Liar')))         ← Stabilized as node
Step 4: 🔁 node(🔁 node(...))                         ← Keeps wrapping in nodes
```

Starting point: named "Liar" self_ref
State: Paradox (self-referential statement)

Demo 1: Liar Paradox (`named "Liar" self_ref`)

```
Step 0: 🏷️ 'Liar'                                    ← Start: paradoxical term
Step 1: 🔥 cap(🏷️ 'Liar')                            ← Encapsulated (paradox wrapped)
Step 2: 🌱 enter(🔥 cap(🏷️ 'Liar'))                  ← Reentering
Step 3: 🔁 node(🌱 enter(🔥 cap(🏷️ 'Liar')))         ← Stabilized as node
Step 4: 🔁 node(🔁 node(...))                        ← Keeps wrapping in nodes
```

Transformation: Paradox detected → Encapsulated

Rule applied: classify(liar) = Paradox → step(liar) = cap(liar)

Demo 1: Liar Paradox (`named "Liar" self_ref`)

```
Step 0: 🏷️ 'Liar'                                    ← Start: paradoxical term
Step 1: 🔥 cap(🏷️ 'Liar')                             ← Encapsulated (paradox wrapped)
Step 2: 🌱 enter(🔥 cap(🏷️ 'Liar'))                    ← Reentering
Step 3: 🔁 node(🌱 enter(🔥 cap(🏷️ 'Liar')))            ← Stabilized as node
Step 4: 🔁 node(🔁 node(...))                          ← Keeps wrapping in nodes
```

Transformation: Capsule reenters the system

Rule applied: classify(cap(liar)) = Integrate → step = enter(cap(liar))

Demo 1: Liar Paradox (`named "Liar" self_ref`)

```
Step 0: 🏷️ 'Liar'                                    ← Start: paradoxical term
Step 1: 🔥 cap(🏷️ 'Liar')                            ← Encapsulated (paradox wrapped)
Step 2: 🌱 enter(🔥 cap(🏷️ 'Liar'))                  ← Reentering
Step 3: 🔁 node(🌱 enter(🔥 cap(🏷️ 'Liar')))         ← Stabilized as node
Step 4: 🔁 node(🔁 node(...))                        ← Keeps wrapping in nodes
```

The Liar paradox goes through the **3-step cycle**: paradox → cap → enter → node, then continues wrapping in more nodes (stable state).

Demo 1: Liar Paradox (`named "Liar" self_ref`)

```
Step 0: 🏷️ 'Liar'                                     ← Start: paradoxical term
Step 1: 🔥 cap(🏷️ 'Liar')                             ← Encapsulated (paradox wrapped)
Step 2: 🌱 enter(🔥 cap(🏷️ 'Liar'))                   ← Reentering
Step 3: 🔁 node(🌱 enter(🔥 cap(🏷️ 'Liar')))          ← Stabilized as node
Step 4: 🔁 node(🔁 node(...))                          ← Keeps wrapping in nodes
```

Transformation: Stabilizes as a node

Rule applied: classify(enter(cap(liar))) = Reentry → step = node(...)

Demo 2: Nested paradox processes outer-first (fuel = 4) - Line 39

```
Step 0: 🏷️ 'Outer'                                  ← named "Outer" (named "Inner" self_ref)
Step 1: 🔥 cap(🏷️ 'Outer')                           ← System detects the OUTER label
Step 2: 🌱 enter(🔥 cap(🏷️ 'Outer'))
Step 3: 🔁 node(🌱 enter(🔥 cap(🏷️ 'Outer')))
Step 4: 🔁 node(🔁 node(...))                        ← Now in stable "node wrapping" phase
```

Key insight:
Even though there's a nested `named "Inner" self_ref` inside, the system processes the **outermost name first**.
Step 4 is extra: After reaching `node(...)`, any further steps just keep wrapping in more nodes (stable absorption).

The 3-Step Cycle as Structural Invariant

1. Detect paradox → 🔥 Encapsulate (cap)
2. Encapsulated paradox → 🌱 Reentry (enter)
3. Reentering capsule → 🔁 Stabilize (node)

Key Properties Demonstrated

- Deterministic: Always the same transformation
- Controlled: No logical explosion
- Universal: Works for simple, named, and nested paradoxes
- Verified: Assertions prove the expected behavior

Formal Verification

Completion (Examples.lean)

These use `completeFuel` which tries to resolve ALL paradoxes recursively in one go (not step-by-step).

Completion Example 1: Liar

```
Before: named 'Liar' (self_ref)                         ← Original paradox
After:  node(enter(cap(named 'Liar' (self_ref))))       ← Wrapped once
Contains paradox (before): true                         ← Has self_ref
Contains paradox (after): true                      ← STILL has self_ref inside!
```

Why still paradox? The `completeFuel` function wraps the *entire term* once as `node(enter(cap(...)))`, but the inner `self_ref` is still there. The wrapping doesn't eliminate the paradox, it just "contains" it.

Completion Example 2: Nested Paradox (`cap(self_ref)`)

```
Before: cap(self_ref)                          ← Paradox already inside cap
After:  cap(node(enter(cap(self_ref))))        ← Inner self_ref resolved
Contains paradox (before): true
Contains paradox (after): true                 ← Outer cap still wraps a paradox
```

What happened? `completeFuel` detected `self_ref` inside the `cap`, wrapped it as `node(enter(cap(self_ref)))`, then put that back inside the outer `cap`.
The structure changed but `self_ref` is still there.

Key Insights

1. **step (step-by-step)**: Processes the **outermost** paradox one step at a time
    - Paradox → cap → enter → node → (keeps wrapping in nodes)
2. **completeFuel (recursive completion)**: Tries to resolve **all** paradoxes at once
    - Wraps paradoxes as node(enter(cap(...)))
    - BUT: self_ref itself never disappears - it's the atomic paradox
    - The wrapping just "stabilizes" the structure

Key Insights

3. Why "Contains paradox: true" after completion?
    - self_ref is the **irreducible paradox** - you can't eliminate it
    - You can only wrap/contain it in stable structure
    - contains_paradox checks if self_ref appears **anywhere** in the tree
    - Even after wrapping, self_ref is still there (just deeper in the tree)
4. **This is by design**: The Gödel Mirror doesn't "solve" paradoxes, it **metabolizes** them into stable structures that don't explode the logic. The paradox is still there, but controlled.

Three Proven Theorems

- **Progress:** Every non-value term can take a computational `step`. The system never gets stuck.
  - For any term t ∈ MirrorSystem, either t is a value (i.e., t = base) or there exists a term t′ such that t → t′.
- **Non-Explosion:** Any paradoxical term resolves deterministically to a stable `node` in exactly **3 steps**.
  - t → cap(t) → enter(t) → node(t)
- **Label Preservation:** A `named` structure keeps its label throughout the reduction process.

For proofs, can see MetaTheory.lean in repo.

Takeaways

Why Lean 4?

Lean provided a complete toolkit:

1. **Precision:** Every rule and definition is type-checked. No ambiguity.
2. **Execution:** The fuelled evaluator allowed for rapid testing and live demos.
3. **Proofs:** I could formally verify critical properties like non-explosion.

Without Lean, the Gödel Mirror would be a conceptual sketch. **With Lean, it is a verifiable system.**

What Was Hard

The Challenge: Non-Termination

Lean's functions must be **terminating** (total) to ensure logical consistency. But the Gödel Mirror is deliberately cyclic and non-terminating.

The Solution: A Two-Pronged Approach

- **For Demo:** Use a **fuelled evaluator** (run n). This is a total function that simply stops after n steps, making it easy to execute.
- **For Proof:** Define the semantics as a **relation** (step : MirrorSystem → MirrorSystem → Prop). This allows reasoning about infinite computations without writing non-terminating functions. (still work-in-progress)

Lean's flexibility handled both the executable spec and the meta-theory.

Why It Matters for Lean Community

Takeaways from Gödel Mirror	Takeaways for Lean Community
Modeling Non-Terminating Calculi Lean can formalize systems that go beyond strong normalization.	**Dedicated TRS Library Needed** Isabelle/HOL has a mature rewriting library; Lean doesn’t yet.
Fuel + Relational Semantics Pattern Combining executable fuel evaluators with relational proofs is a practical technique for non-terminating semantics.	**Automation:** Avoid reproving confluence/termination lemmas manually. **Accessibility:** Lower barrier for CS researchers and students. **Advanced Topics:** Enables systematic work on infinitary and non-terminating rewriting.
Lean as Executable Spec Lab Beyond theorem proving, Lean works as a design studio for new formal systems, where you can prototype, run, *and* prove in one place.	**Path Forward** Like mathlib for mathematicians. It’d be great a TRS library for PL/rewriting researchers too.

Conclusion

1. Contradictions don't have to explode, they can be **metabolized** into structured computational nodes.

   Paradox → cap → enter → node

2. **Lean 4 made it real**. The only reason I could test, run, and prove this as an outsider was because Lean integrates precision, execution, and proof in one place.

Gödel Mirror

A Paraconsistent Calculus that Metabolizes Contradictions Mechanized in Lean 4

Demo repo (branch ‘itp’):
https://github.com/jhetchan/godel-mirror/tree/itp

Jhet Chan

https://jhetchan.com

Independent Researcher, Malaysia

Questions