Lean for Science Formalization

Przemyslaw Chojecki

ulam.ai

February 18, 2026

Abstract

Lean has matured from a mathematics-first proof assistant into a plausible *scientific* formalization substrate: a small-kernel, dependently typed language with fast metaprogramming and a large shared library (`mathlib`) that can host domain libraries, executable reference implementations, and verified interfaces to external solvers. Here we synthesize the emerging evidence across physics, chemistry, biology, and economics, highlighting concrete Lean 4 artifacts—PhysLean for "digitalizing" physics (including index notation and Wick's theorem), Lean mechanizations of classical chemical derivations (Langmuir and BET adsorption), proof-backed molecular energy computation benchmarked to NIST reference data, semantics-heavy biological modeling via the continuous π-calculus, and mechanized economic results spanning voting theory and automated market makers (AMMs). We also connect this agenda to AI-driven chip design (EDA).

We then propose a *mathematics-first transfer strategy* for **autoformalization beyond mathematics**: train and validate autoformalization agents on mathematics (where verification is cleanest and corpora are largest), then port the same agentic loop to science via domain-specific notations, typed invariants (units, tensor shapes, state transitions), and certified numerical bridges. Finally, we outline how an agentic stack such as `ulamai` can operationalize this workflow end-to-end, turning scientific texts, datasets, and benchmarks into formally checked libraries and executable, audited scientific kernels.

Introduction

Formal verification is no longer a niche activity confined to software correctness or foundational mathematics. The last two years have seen a growing set of Lean 4 artifacts that target *scientific* knowledge: physics libraries that respect physicist-native notation, mechanized reconstructions of canonical chemistry derivations, verified scientific computation that matches external benchmarks, and economic models where proofs directly support auditing and security (e.g., DeFi market mechanisms). These developments matter because scientific correctness failures often arise from *hidden assumptions*, *implicit invariants*, and *boundary mistakes* (units, indexing conventions, state transitions, rounding errors)—precisely the kinds of obligations proof assistants force into the open.

Lean is technically well-positioned for this shift. Lean 4 is a reimplementation aimed at extensibility, performance, and ecosystem-scale development [3, 1]. Its build and toolchain workflow (Lake + Elan) supports reproducible, dependency-pinned scientific libraries [5, 4, 1], while documentation tooling (doc-gen4) enables "scientific knowledge base" presentation comparable to mathlib documentation [6]. The result is an ecosystem where scientific domain libraries can be treated as *versioned, buildable, reviewable artifacts*.

Other proof assistants have long track records in mathematics and engineering, including scientifically flavored formalizations. Isabelle/HOL combines interactive proof with strong automation and "hammer"-style ATP/SMT integration (Sledgehammer) [52, 53], and it has hosted verified numerical analysis developments such as mechanized ODE theory and one-step methods with explicit error bounds [54]. Coq has a deep ecosystem for certified programming and arithmetic, including efficient, formally verified floating-point computation inside proofs [55]. HOL Light similarly emphasizes a lightweight, LCF-style kernel and has been applied to substantial formalization work in real analysis and floating-point verification [56]. We focus on Lean because Lean 4's macro system and metaprogramming make domain-native notation layers practical, `mathlib` provides a broad shared mathematical substrate, and the ecosystem now exposes AI-facing interfaces for data extraction/orchestration (LeanDojo, LeanInteract) and solver-backed automation with proof replay (Lean-SMT) [3, 7, 26, 27, 25].

The open question is not whether Lean can express scientific models, but *how* it should be used to maximize scientific value. Existing artifacts suggest a consistent pattern: Lean is most effective as a *formal core* that (i) verifies derivations and invariants, (ii) makes assumptions explicit, and (iii) supports reusable domain libraries—while treating high-performance numerics and data as carefully controlled boundaries. We therefore frame Lean for science as a compositional engineering problem: build domain kernels (units, tensors, state models), connect them to mathematics (mathlib), connect to computation (SciLean, LeanCert), and connect to automation (SMT/ATP, retrieval agents).

We do not claim that Lean replaces scientific computing stacks, ML training frameworks, or production EDA toolchains; these systems exist to optimize performance, throughput, and hardware utilization rather than to serve as proof objects. Our claim is narrower: Lean is valu-

able as a *formal core* for certifying invariants, semantics, and trust boundaries around models, derivations, and tool-driven pipelines, especially where failures arise from implicit assumptions or mismatched interfaces. In practice, many components will remain approximate or externally implemented (numerics, simulators, solvers, EDA flows); the goal is to make those boundaries explicit and to attach checkable certificates and provenance metadata wherever feasible.

Lean as a scientific formalization substrate

Lean's design aligns with scientific formalization because it treats specifications, proofs, and programs as first-class, checkable objects [1, 2]. Lean 4's extensibility (macros, syntax, metaprogramming) enables domain-native notations and workflows rather than forcing scientists into unnatural encodings [3]. In practice, three ecosystem ingredients are decisive:

(i) Library substrate. `mathlib` provides a broad base of algebra, analysis, probability, and category theory that domain libraries can import rather than re-prove [7]. This is a multiplier for physics (analysis/manifolds), chemistry (series, thermodynamics abstractions), biology (probability foundations for stochastic models), and economics (fixed points, order theory).

(ii) Project reproducibility. Lake (build/package manager) and Elan (toolchain manager) standardize dependency resolution and per-project version pinning [5, 4, 1]. Scientific formalization is unusually sensitive to bitrot; pinned toolchains and CI workflows are therefore not optional engineering details.

(iii) Interfaces for AI and automation. Tooling increasingly treats Lean repositories as structured data sources and interactive environments. LeanDojo offers repository tracing, dataset generation, and agent scaffolding for Lean 4 [26], while LeanInteract exposes a Python interface to Lean 4 via the REPL for orchestration in scientific pipelines [27]. Solver integration is also becoming practical: Lean-SMT translates goals to SMT queries (e.g., cvc5) and replays solver proofs inside Lean [25].

Evidence base across scientific domains

Table 1 summarizes a selection of *Lean-first* scientific artifacts that move beyond "toy" examples.

Physics: building libraries and notations

PhysLean explicitly targets a unified physics library in Lean 4, organized by topics and accompanied by documentation [8, 9]. Its scope is notable: electromagnetism, quantum mechanics, statistical mechanics, relativity, particle physics, and quantum field theory. Beyond breadth, PhysLean supports *domain-native notation*. A key example is the formalization of physics index notation in Lean 4, designed to bridge tensor notation and formal verification, with category-theoretic structure supporting the implementation [10]. Another milestone is the digitalization of Wick's theorem and variants in Lean 4 [11]. Together, these efforts suggest that Lean can host not only physics theorems but the *linguistic infrastructure* physicists use to reason.

Chemistry: derivation hygiene and verified computation

Chemistry currently shows two strong Lean pathways. First, mechanized reconstructions of standard derivations in adsorption theory (Langmuir and BET) demonstrate how Lean forces explicit premises and reveals missing side-conditions [13, 14]. Second, LeanLJ provides a more computation-oriented result: a Lean 4 package implementing Lennard–Jones energy calculations under periodic boundary conditions, with correctness arguments and agreement with NIST reference calculations [15, 16]. LeanLJ's methodology is especially instructive: proofs over idealized $\mathbb{R}$ coexist with executable code over floating point via polymorphism and typeclasses, and IO is treated as an explicit boundary rather than contaminating the pure core [15].

Biology: semantics first, discrete invariants next

Lean-native biology is thinner than physics and chemistry, but credible directions exist. The mechanization of the continuous π-calculus targets biochemical process modeling with formal syntax and semantics, and demonstrates generation of ODE-style semantics from process descriptions [18, 19, 17]. This aligns with biology's need for reproducible semantics when models are transformed, composed, or parameterized. A complementary "small but concrete" direction is discrete invariants: proofs about properties of the genetic code in Lean 4 treat the code as a finite function and establish basic structural properties [20]. Such invariants can act as certified guardrails inside computational biology pipelines.

Economics: mechanized semantics where proofs are brittle

Economics has several mechanization footholds where Lean's strengths are unusually direct. Voting theory involves subtle quantifier structure and variable-population axioms; a Lean framework explicitly targets this setting using dependent types and applies it to verifying voting method properties [21]. DeFi provides a modern testbed where formally specified economic mechanisms have immediate practical stakes: constant-product AMMs have been formalized in Lean 4 with mechanized proofs of properties such as arbitrage [22]. These cases show how Lean can serve as an auditable semantics layer for socio-economic protocols.

Hardware and EDA: AI-driven chip design as certificate-driven synthesis

Lean-native EDA formalizations are still nascent; we include chip design because it is a clean "specification + optimization" domain where certificate-driven checking and explicit trust boundaries are unusually well motivated.

AI-driven chip design provides a particularly instructive adjacent domain for "Lean beyond mathematics". Modern EDA workflows are already *specifications plus optimization*: netlists, timing constraints, and geometry rules define a large constraint system; solvers and heuristics search for a layout that satisfies those constraints while optimizing PPA. Recent work pushes this further by using learning to propose *design artifacts* directly, notably deep RL for macro placement/floorplanning and open-source replications intended for practical use [38, 39]. In parallel, ML research explores more tightly coupled placement–routing learning and neural routing generation, aiming to reduce hand-crafted heuristics and improve scalability [44, 45]. Survey work indicates that ML now touches many stages of EDA, from prediction (QoR, congestion, timing) to decision-making (synthesis, placement, routing) [37].

However, EDA also highlights why verifier feedback must go beyond "compiles" or "type-checks": the dominant failure mode is not syntactic invalidity but *metric and constraint mis-specification*. Independent assessments of learning-based macro placement stress missing methodological details, baseline sensitivity, and the fragility of claims when evaluated under transparent implementations and broader test suites [40, 41]. More recent benchmarking work explicitly targets the gap between intermediate surrogate metrics and *end-to-end* PPA measured after running a full flow, showing that strong proxy performance can still yield unsatisfactory final outcomes [42]. This pattern closely mirrors the "hidden assumptions" theme in scientific derivations: in EDA, the hidden assumptions are often embedded in tool defaults, proxy costs, and unrecorded constraints.

From the Lean perspective, chip design is a natural instance of *certificate-driven constrained synthesis*: treat ML/RL/LLM systems as *proposal generators* and treat Lean as the *acceptance filter* that enforces domain validity predicates (DRC-style geometric legality, connectivity equivalence, timing constraints, state invariants for flows). Open-source infrastructure such as OpenROAD makes it feasible to integrate this loop end-to-end and to standardize artifacts for evaluation [43]. In addition, LLM-driven orchestration of open-source RTL-to-GDSII pipelines (e.g., MCP-style tool control and backend-aware optimization loops) shows that the orchestration layer can be automated; Lean-style certificates provide a principled trust boundary for such automation [46]. Finally, existing Lean artifacts translating large hardware specifications (e.g., Sail RISC-V ISA semantics) indicate that Lean can already host the semantics needed to connect "chip design outputs" to correctness claims at the architecture and toolchain levels [47].

Machine learning and LLMs: from formal semantics to certified transformers

Machine learning is a natural "Lean beyond mathematics" target because it mixes math, code, and empirical claims. Recent work shows Lean can host *model semantics* and *learning theory* as checkable artifacts: a Lean 4 development formalizes Hopfield networks and Boltzmann machines, proving convergence and (restricted) Hebbian-learning correctness, and establishing ergodicity for Boltzmann-machine dynamics [48]. Complementarily, learning-theoretic infrastructure is being mechanized in Lean 4, including generalization bounds via Rademacher complexity and broader empirical-process foundations for statistical learning theory [49, 50]. These efforts directly support "assumption hygiene" in ML: side conditions and measurability/regularity requirements become explicit obligations rather than informal footnotes.

For modern LLM workflows, the most credible near-term role for Lean is *certificate-driven trust boundaries*: untrusted tools propose approximations, edits, or extracted mechanisms, while Lean checks composition logic and provenance. BlockCert exemplifies this pattern by attaching machine-checkable per-block certificates to transformer surrogates and formalizing in Lean 4 a simple composition theorem that lifts local error guarantees to a global deviation bound (under explicit assumptions), with empirical application to real transformer checkpoints [51]. In parallel, projects such as SciLean indicate a path toward shape-safe differentiable kernels (arrays + AD) inside Lean [23]. Together, these directions suggest a practical architecture-discovery loop: search proposes candidate modules or edits, domain validity predicates enforce invariants, and Lean-certified certificates record what is truly guaranteed versus merely observed.

Design patterns emerging from current artifacts

Across domains, several engineering patterns recur.

Explicit assumptions as reusable structures. Chemistry formalizations emphasize that derivations become clearer when assumptions are packaged and reused rather than sprinkled informally [13]. This generalizes: in economics, "axiom bundles" for voting methods; in physics, consistent dimension systems and tensor index constraints [12, 10].

Domain kernels before grand unification. PhysLean succeeds by building a monolithic-but-modular physics library and inviting incremental contributions [8]. Chemistry succeeds by formalizing canonical derivations and extracting reusable thermodynamics/kinematics scaffolding [13]. This suggests that near-term progress comes from small, reusable domain kernels (units, tensors, state machines) rather than attempting end-to-end scientific stacks in one step.

Trust boundaries for numerics and data. Verified computation projects isolate IO and approximation, proving theorems about pure cores while treating external data ingestion explicitly [15]. LeanCert extends this idea to verified bounds via rigorous interval arithmetic and Taylor models, producing proof objects checked by Lean's kernel [24].

Solver-backed automation with proof replay. LeanSMT demonstrates a practical hybrid: discharge certain goals to SMT solvers and replay (parts of) proofs inside Lean [25]. This can scale reasoning in domains like eco-

nomics and combinatorial biology, where solver search is powerful but trust must be restored via certificates.

Mathematics-first autoformalization as a transfer strategy

Scientific autoformalization is harder than mathematical autoformalization: scientific texts mix informal modeling assumptions, units, approximations, and domain jargon, and frequently depend on empirical conventions rather than purely axiomatic foundations. A pragmatic strategy is therefore **mathematics first, science second**.

Why start with mathematics?

Mathematics provides (i) large corpora of formal content (`mathlib`) [7], (ii) clean verifier feedback with minimal modeling ambiguity, and (iii) well-defined evaluation targets. Tooling such as LeanDojo turns Lean repositories into structured training data and supports retrieval-augmented theorem-proving agents [26]. Autoformalization benchmarks such as LeanEuclid stress statement translation and proof search in a controlled geometry subdomain [28]. LeanAide illustrates practical "AI-in-the-loop" workflows centered on autoformalization and tool support within Lean projects [29].

A canonical agentic loop

A general-purpose autoformalization loop can be expressed as a verifier-driven search process:

1. **Retrieve context:** select relevant definitions/lemmas from `mathlib` and domain libraries (retrieval over declarations, docstrings, dependency graphs) [26].
2. **Propose formalization:** generate Lean statements (and optionally tactic scripts) for the target claim.
3. **Check and repair:** run Lean elaboration; use error messages to repair syntax/types; iterate.
4. **Prove or decompose:** if direct proof fails, propose sublemmas; repeat with verifier feedback.
5. **Package:** export successful artifacts as library code, with documentation and CI builds (Lake/Elan) [5, 4, 6].

The key point is that Lean supplies a *dense reward signal*: every type error, missing lemma, or proof gap is a concrete gradient for iterative refinement. In mathematics, this loop can be trained and evaluated at scale.

Transferring to science: what changes?

Transferring the loop to science requires representing non-mathematical structure in types, definitions, and invariants. Existing science artifacts already provide prototypes of the required "representation upgrades": (i) *domain notation layers* (index notation [10]); (ii) *cross-domain invariants* (dimensions/units [12]); (iii) *state-transition semantics* (AMMs [22], process calculi [18]); (iv) *spec-vs-exec bridges* for numerics (LeanLJ [15], LeanCert [24]); (v) *solver certificates* (Lean-SMT [25]).

A further "engineering transfer" case is EDA: learning-based floorplanning/placement illustrates a realistic propose–then–check pipeline, while open reassessments and end-to-end benchmarks motivate explicit validity predicates and trust accounting rather than proxy-metric optimism [38, 39, 40, 42].

Thus, "science autoformalization" is best viewed not as translating prose directly into full physics or biology, but as translating scientific knowledge into *layered* Lean artifacts: a vocabulary layer, an assumption layer, a theory layer, and (optionally) a certified computation layer.

Beyond mathematics: how ulamai-style autoformalization could work

We now sketch how an agentic stack (e.g., `ulamai` [32]) can implement the above loop beyond mathematics. We emphasize *architecture and interfaces* rather than claiming a specific implementation, because scientific domains will require iterative co-design with domain experts.

Start from mathematical benchmarks and datasets

A mathematics-first curriculum can be grounded in open-problem datasets that reward research-like behavior and formal verification. The UnsolvedMath dataset and its accompanying benchmark framing propose evaluating literature search, theorem proving, proof checking, and formalization as distinct axes [30, 31]. This perspective generalizes naturally to science: scientific progress often consists of verified *partial results*, repaired derivations, and clarified assumptions rather than complete "final answers."

Extend the loop with domain adapters

To move beyond mathematics, the agent needs domain adapters that translate scientific artifacts into Lean layers:

Adapter A: Notation and invariants. For physics and chemistry, integrate dimensional analysis as a default typing discipline [12], and provide tensor/index DSLs where needed [10]. For economics and computation-heavy domains, add solver-backed tactics for arithmetic/optimization subgoals [25].

Adapter B: Model semantics. For biology, represent process languages, protocols, or reaction networks as inductive objects with operational semantics [18]. For economics/DeFi, represent markets as state-transition systems with typed invariants and economic observables [22].

Adapter C: Certified numerics bridges. When scientific claims require numeric evidence, use a two-type strategy: prove over $\mathbb{R}$ and run over floating point with certified bounds or interval arithmetic [15, 24]. In domains where arrays and differentiation are central, SciLean provides early infrastructure for numerical computing and AD in Lean 4 [23].

Adapter D: Orchestration and data plumbing. Use programmatic interfaces to run Lean from Python for dataset construction, regression testing, and evaluation pipelines [27, 26]. Keep IO explicit and confined.

Adapter E: Hardware/EDA design automation. For chip design, represent netlists and layout primitives as typed graphs/geometry with explicit invariants (connectivity, legality, timing constraints), treat EDA tools as untrusted oracles, and import their outputs as certificates to be checked (or replayed) by Lean. Open-source flows such as OpenROAD provide an evaluation substrate and standardized artifacts for end-to-end measurement [43], while LLM-controlled orchestration systems demonstrate how agents can run closed-loop RTL-to-GDSII optimization with real backend feedback [46]. At the specification boundary, Lean-hosted ISA semantics (e.g., Sail RISC-V translated to Lean) can anchor correctness claims across microarchitecture, compilation, and verification pipelines [47].

A concrete end-to-end workflow

A realistic "paper-to-Lean" scientific pipeline can be staged:

1. **Blueprint extraction:** from a scientific document, extract definitions, assumptions, target claims, and the dependency graph of lemmas (informal blueprint).
2. **Lean skeleton generation:** generate Lean `structures` for assumptions and definitions; encode invariants (units, indices, state well-formedness).
3. **Verifier-guided completion:** iteratively elaborate and discharge goals, using retrieval and solver tactics where applicable [26, 25].
4. **Executable kernel (optional):** implement a reference computation (e.g., energy kernel, pricing function, protocol transition) and prove key properties; validate against external benchmarks [15, 16].
5. **Packaging:** publish as a Lake project with pinned toolchain and docs generated via doc-gen4 [5, 4, 6].

This workflow mirrors what has already happened in chemistry (derivation + computation) [13, 15] and economics (formal mechanism + economic properties) [22], suggesting that science autoformalization can proceed incrementally by targeting invariant-rich kernels.

What changed, and why it matters now

Several shifts explain why "Lean for science" is no longer speculative.

Lean 4 ecosystem maturity. Lean 4's extensibility and performance goals were designed to support large libraries and domain-specific tooling [3, 1]. Lake/Elan standardize reproducible builds [5, 4], and doc-gen4 provides documentation workflows needed for scientific adoption [6].

Domain libraries have appeared. PhysLean has moved physics formalization toward a library-building posture with explicit entry points and associated publications [8, 9]. Chemistry has produced both derivation formalizations and verified computational kernels benchmarked to external references [13, 15, 16]. Economics has produced Lean 4 formalizations in DeFi with mechanized economic theorems [22]. Adjacent engineering and ML-facing artifacts are also emerging: large hardware semantics libraries (Sail RISC-V translated to Lean) and early certificate patterns for transformer components suggest that "small-kernel checking + untrusted generators" can apply beyond traditional scientific domains [47, 51].

Autoformalization tooling is becoming practical. Dataset extraction and agent frameworks (LeanDojo), interactive Python orchestration (LeanInteract), and in-editor AI tooling (LeanAide) lower the barrier to building verifier-guided loops [26, 27, 29]. Solver tactics with proof replay (Lean-SMT) extend automation to numerically flavored subgoals [25].

What can be researched next? A prioritized agenda

The landscape suggests an actionable research program.

1. Domain kernels as shared infrastructure

Prioritize small, reusable kernels: *units/dimensions* (already formalized) [12]; *tensor/index DSLs* [10]; *market/state semantics templates* [22]; *process-language semantics templates* [18]. The goal is to make most scientific formalizations "start from imports" rather than from scratch.

2. Certified numerics patterns

Extract LeanLJ's spec/exec bridge into a general template for scientific kernels [15]. Combine this with LeanCert-style certified bounds for robust guarantees [24]. A unifying interface for "approximate computation with proof of error" would directly benefit physics (PDE/ODE approximations), chemistry (simulation kernels), biology (stochastic models), and economics (numerical equilibria).

3. Autoformalization datasets for science

Mathematics-first training is necessary but not sufficient. Build scientific autoformalization datasets by curating: (i) derivation corpora (adsorption, thermodynamics, mechanics) [13]; (ii) domain library tasks (PhysLean goals; AMM lemmas) [8, 22]; (iii) hybrid tasks requiring both formal proofs and benchmark conformance [15, 16]. Leverage UnsolvedMath-style evaluation dimensions (search, proof, check, formalize) as a general rubric [31].

4. Human–AI collaboration protocols

PhysLean explicitly invites informal contributions as a pathway for domain experts [8]. Generalize this pattern: separate roles (domain expert, formalizer, reviewer), and treat informal blueprints as first-class artifacts that autoformalization agents can target. Tooling such as doc-gen4

can turn evolving libraries into navigable scientific references [6].

5. Hard frontier: PDEs, inference, and empirical pipelines

Many real scientific models are PDE-centric, stochastic, and data-calibrated. The critical next steps are: (i) PDE formalization and certified numerical PDE solvers in Lean; (ii) probabilistic inference workflows that connect measure-theoretic foundations to executable sampling with correctness guarantees; (iii) typed, auditable data interfaces that preserve provenance and assumptions end-to-end. These are large undertakings, but the emerging pieces (SciLean for arrays/AD [23], LeanCert for bounds [24], Lean-SMT for automation [25]) suggest a plausible convergence path.

SciAF: a benchmark suite for scientific autoformalization

Motivation and scope

Mathematics-first autoformalization provides clean verification signals and large corpora, but scientific texts introduce two extra burdens: (i) *domain validity* (units, shapes, state invariants, semantic well-formedness), and (ii) *approximation boundaries* (numerics, data, empirical conventions). To make progress measurable, we propose **SciAF** (*Scientific AutoFormalization*): a benchmark suite designed to evaluate autoformalization *beyond* mathematics by combining Lean elaboration with domain-specific validity checks and trust accounting. SciAF is designed to be executed by an agentic orchestration layer; we use `ulamai` as a reference implementation scaffold for these loops and evaluation harnesses [32].

SciAF is seeded by existing Lean-first scientific artifacts and their accompanying informal sources: physics libraries and notation layers (PhysLean, index notation, dimensional analysis) [8, 10, 12], chemistry derivations and verified kernels (Langmuir/BET, LeanLJ) [13, 15], biology semantics formalizations (continuous π-calculus) [18], and economics mechanisms (voting, AMMs) [21, 22]. We also include adjacent engineering domains where "propose–then–check" is the native workflow, notably hardware/EDA artifacts (Sail RISC-V ISA semantics; open-source flows such as OpenROAD; LLM-driven RTL-to-GDSII orchestration) [47, 43, 46] and early Lean-touching ML/LLM certification work (Hopfield/Boltzmann networks in Lean; transformer block certificates) [48, 51].

Task suite

Each SciAF item contains: (a) a natural-language claim (and optionally a derivation sketch), (b) a declared Lean environment header (`imports` + namespaces), and (c) optional executable references (datasets/benchmarks) when the task is computation-facing. We define four primary tasks, with two science-specific extensions:

From type-check filtering to domain validity filtering

In mathematical autoformalization, a dominant failure mode is simply producing statements that do not typecheck. Poiroux *et al.* show that decoding with *type-check filtering* substantially improves statement autoformalization, achieving large absolute accuracy gains on ProofNet in Lean 4 [33, 34]. SciAF generalizes this idea: **typechecking is necessary but not sufficient** for science. A statement can elaborate while being scientifically invalid (dimensionally inconsistent equations, ill-scoped tensor indices, invalid market states, or illegal process transitions).

We therefore introduce **domain validity predicates** (DVPs): Lean-encoded predicates that capture domain well-formedness constraints and act as an additional verifier layer. Examples include:

- **Units and dimensions:** `DimHomogeneous eqn` (cross-domain) [12];
- **Tensor hygiene:** index-range and contraction well-formedness [10];
- **State invariants:** `ValidAMMState s` in market mechanisms [22];
- **Semantic well-formedness:** well-scoped names/binders in biochemical process terms [18];
- **EDA legality/connectivity:** design-rule legality predicates and netlist connectivity equivalence checks (e.g., "layout realizes netlist"), motivated by end-to-end open flows [43];
- **ML/LLM structural validity:** shape/mask well-formedness for tensor programs and schema checks for blockwise error certificates [51].

Operationally, SciAF evaluates candidates by the conjunction

$$\mathrm{Accept}(c) \;:=\; \mathrm{Elab}(c) \;\wedge\; \mathrm{DVP}(c), \tag{1}$$

where $\mathrm{Elab}(c)$ denotes Lean elaboration success and $\mathrm{DVP}(c)$ denotes domain validity checks for the target track.

Metrics

SciAF uses metrics that distinguish (i) compilation, (ii) scientific validity, (iii) proof provenance, and (iv) executable conformance:

- **Elaboration rate** Elab_k: fraction of problems for which at least one of k samples elaborates.
- **Validity rate** DVP_k: fraction of problems for which at least one of k samples satisfies $\mathrm{Accept}(\cdot)$.
- **Kernel proof rate** KProof: fraction of problems solved with kernel-checked proofs (no untrusted oracles).

- **Trust budget** TB: vector-valued accounting of reliance on external ATP/SMT without reconstruction, and on numerical claims without certified bounds (interval/Taylor certificates) [25, 35, 24].
- **Assumption completeness rate** ACR: fraction of successful formalizations whose explicit hypotheses suffice for the proof without hidden side conditions.
- **Assumption minimality** AMin: hypothesis-set size (or weakening distance) after ablation-based minimization (drop-one hypothesis tests until failure).
- **Spec–exec refinement coverage** RefCov: fraction of executable components linked to the spec by proved refinement/bound theorems; particularly relevant for SciAF-Exec [15].

Implementation note. SciAF is designed to be run using repository tracing and Lean compilation feedback loops [26, 27]. DVPs are expressed as Lean definitions and evaluated as part of the elaboration-and-proof pipeline; trust labels are emitted as metadata alongside proofs and executables.

Baseline and reproducible execution. A minimal SciAF baseline is: retrieve context from the target Lean environment (declaration graph + nearest-neighbor lemma retrieval), prompt an LLM to propose candidate statements/proofs, then apply *type-check filtering* (elaboration success) [33] followed by SciAF-specific *DVP filtering* (units/shapes/state validity) before scoring. A practical runner is a pinned Lake/Elan project executed via LeanDojo/LeanInteract-style orchestration (batch compilation, tracing, and replayable evaluation) [5, 4, 26, 27].

Pilot study: a hidden-assumption ledger for scientific derivations

Rationale

A recurring discovery in Lean-based chemistry and physics formalizations is that "standard" derivations silently rely on side conditions (e.g., denominators nonzero; positivity; domain restrictions), which become explicit obligations in Lean [13]. We propose a **hidden-assumption ledger** pilot study to quantify and classify these obligations across domains, and to turn them into training/evaluation data for SciAF-Assump.

Protocol (template for a reproducible pilot)

Select a small corpus (e.g., 5 derivations per domain, $n = 20$ total) spanning: (i) symbolic derivations (Langmuir/BET, dimensional analysis exercises), (ii) semantic/state models (AMMs, process calculi), (iii) computation-facing kernels (LeanLJ-style energy, arbitrage calculation). For each item:

1. **Seed formalization:** attempt a minimal Lean statement from the informal claim.
2. **Failure mining:** run elaboration/proof attempts; log Lean error states and remaining goals.
3. **Assumption repair:** iteratively add hypotheses until the statement becomes provable.
4. **Minimization:** ablate hypotheses (drop-one, then drop-many) to approximate a minimal set.
5. **Ledger entry:** record (a) added hypotheses, (b) their category, (c) the proof step they enabled.

This pipeline is naturally implemented as a verifier-guided loop (Lean compilation feedback) and is amenable to automation by agentic tooling [26, 27, 32].

Assumption taxonomy (scientific obligations surfaced by Lean)

Pilot outputs and how they become training signal

The pilot produces: (i) a dataset of *assumption repair pairs* (failed statement $\rightarrow$ repaired statement), (ii) a labeled assumption taxonomy distribution, and (iii) a minimization trace that distinguishes "necessary" from "convenient" hypotheses. These outputs support two learning problems:

- **Assumption completion modeling:** predict missing hypotheses from Lean failure states.
- **Assumption minimality modeling:** avoid "assume everything" repairs by learning ablation-stable hypothesis sets.

From verification to discovery: constrained synthesis as "solving" scientific models

Problem formulation

Scientific "solving" is often a synthesis problem: find a model within a family that satisfies invariants, agrees with constraints, and yields provable properties. We formalize this as **certificate-driven constrained synthesis**. Let $\mathcal{M}$ be a model class encoded in Lean (e.g., reaction networks, AMM variants, constitutive laws), let $\mathcal{C}$ be a set of constraints/invariants, and let $\mathcal{T}$ be a set of target theorems (safety, monotonicity, identifiability, arbitrage properties). The goal is to produce a pair (m, π) such that Lean checks:

$$\mathsf{Lean} \vdash \mathsf{Model}(m) \wedge \bigwedge_{c \in \mathcal{C}} c(m) \wedge \bigwedge_{t \in \mathcal{T}} t(m), \qquad (2)$$

where π is a proof certificate (a proof term or a proof-producing trace) for the conjunction. External search (LLMs, SAT/SMT, numerical optimizers) may propose m, but Lean validates it.

A CEGIS-style loop with Lean certificates

Thin-slice exemplars (science-native synthesis targets)

SciAF enables "solving" tasks that go beyond checking a fixed derivation:

Certificate-driven synthesis loop (CEGIS style).

1. **Specify the search space.** Encode model family $\mathcal{M}$ and invariants $\mathcal{C}$ as Lean definitions (domain kernel + DVPs).
2. **Propose candidate.** Use an untrusted generator (LLM/SMT/SAT/optimizer) to propose $m \in \mathcal{M}$.
3. **Validate structure.** Check Elab(m) and DVP(m); reject if invalid.
4. **Prove targets.** Attempt Lean proofs of $\mathcal{T}$ using tactics, SMT replay, and ATP hammers with reconstruction when available [25, 35].
5. **Refine on failure.** If proofs fail, extract counterexamples or residual obligations (failed goals) as new constraints; iterate.
6. **Emit certificate.** Output (m, π) with trust metadata; for numeric claims, prefer certified bounds (interval/Taylor certificates) [24].

Figure 1: Lean as a scientific "solver" via constrained synthesis. External search proposes; Lean certifies. Failures are informative: residual goals become new constraints, enabling counterexample-guided refinement.

- **Dimensionally consistent law discovery:** synthesize candidate constitutive relations constrained by dimensional homogeneity [12], then prove invariants (e.g., monotonicity regimes).
- **Mechanism design under invariants:** synthesize AMM fee rules or swap constraints that preserve desired economic properties, then certify theorems about arbitrage or net-worth behavior [22].
- **Verified microkernels for simulation:** synthesize optimized but refinement-checked kernels (e.g., neighbor-list variants) whose outputs are provably within certified bounds [15, 24].
- **Combinatorial discovery with solver certificates:** use SAT/SMT to search for counterexamples or impossibility witnesses, then certify the encoding/certificate checker in Lean [36].
- **Chip design as certified synthesis:** synthesize floorplans/placements via ML/RL/LLM proposal generators, then certify legality (design-rule constraints), connectivity equivalence, and timing objectives using a Lean-checked ledger; use open EDA flows and end-to-end benchmarks to prevent proxy-metric overfitting [38, 39, 43, 42].

Evaluation for synthesis (what "solving" means operationally)

We recommend reporting, for each synthesis benchmark: (i) success rate of producing *certified* (m, π) pairs, (ii) trust budget TB (kernel-only vs externally trusted), (iii) model complexity (size of m), and (iv) constraint-tightening trajectory (number of refinement rounds). This reframes scientific progress as a machine-checkable sequence of model improvements over an explicit assumption lattice: strengthening or weakening constraints corresponds to moving within a formally represented hypothesis space.

Conclusion

Lean is already supporting meaningful mechanized reasoning in physics, chemistry, biology-adjacent semantics, and economics. The current value proposition is clearest when Lean is used to (i) formalize derivations and invariants, (ii) make hidden assumptions explicit, and (iii) build reusable domain libraries that interoperate with mathlib and certified computation.

Autoformalization provides a credible path to scaling these efforts, but it should proceed as a curriculum: learn to translate and prove mathematics first, then transfer to science via domain adapters for notation, invariants, semantics, and certified numerics. An agentic stack such as `ulamai` [32] can serve as the orchestration layer that turns scientific texts and benchmarks into reproducible Lean artifacts. The next research frontier is therefore not a single "killer app," but a shared infrastructure program: domain kernels, certified numerics bridges, solver-proof pipelines, and datasets that make scientific autoformalization measurable. The same certificate-driven pattern extends naturally to adjacent engineering domains (EDA) and to ML/LLM workflows, where untrusted generators propose artifacts and Lean enforces invariant-rich acceptance filters.

References

[1] Lean documentation. *Extended setup notes (Lean 4)*. https://docs.lean-lang.org/lean4/doc/setup.html (accessed 2026-02-15).

[2] Lean 4 community site. *Lean 4: Learn Functional Programming & Theorem Proving*. https://lean4.dev/ (accessed 2026-02-15).

[3] de Moura, L. & Ullrich, S. *The Lean 4 Theorem Prover and Programming Language*. CADE 28 (LNCS), 2021. DOI: 10.1007/978-3-030-79876-5_37. (Metadata: https://ouci.dntb.gov.ua/en/works/4bYYEea7/, accessed 2026-02-15).

[4] Lean developers. *elan: Lean version manager*. https://github.com/leanprover/elan (accessed 2026-02-15).

[5] Lean developers. *Lake: Lean 4 build system and package manager*. https://github.com/leanprover/lake (accessed 2026-02-15).

[6] Lean developers. *doc-gen4: Document Generator for Lean 4*. https://github.com/leanprover/doc-gen4 (accessed 2026-02-15).

[7] leanprover-community. *mathlib4*. https://github.com/leanprover-community/mathlib4 (accessed 2026-02-15).

[8] HEPLean. *PhysLean: digitalising results from physics into Lean 4*. https://github.com/HEPLean/PhysLean (accessed 2026-02-15).

[9] Tooby-Smith, J. *HepLean: Digitalising high energy physics*. arXiv:2405.08863 (2024). https://arxiv.org/abs/2405.08863 (accessed 2026-02-15).

[10] Tooby-Smith, J. *Formalization of physics index notation in Lean 4*. arXiv:2411.07667 (2024). https://arxiv.org/abs/2411.07667 (accessed 2026-02-15).

[11] Tooby-Smith, J. *Digitalizing Wick's theorem*. arXiv:2505.07939 (2025). https://arxiv.org/abs/2505.07939 (accessed 2026-02-15).

[12] Bobbin, M. P., Jones, C., Velkey, J., & Josephson, T. R. *Formalizing dimensional analysis using the Lean theorem prover*. arXiv:2509.13142 (2025). http://arxiv.org/abs/2509.13142 (accessed 2026-02-15).

[13] Bobbin, M. P. et al. *Formalizing chemical physics using the Lean theorem prover*. *Digital Discovery* **3**, 264–280 (2024). DOI: 10.1039/D3DD00077J. https://pubs.rsc.org/en/content/articlelanding/2024/dd/d3dd00077j (accessed 2026-02-15).

[14] Bobbin, M. P. et al. *Formalizing Chemical Theory using the Lean Theorem Prover*. arXiv:2210.12150 (preprint, 2022; updated versions). https://www.emergentmind.com/articles/2210.12150 (accessed 2026-02-15).

[15] Ugwuanyi, E. D., Jones, C., Velkey, J., & Josephson, T. R. *Benchmarking Energy Calculations Using Formal Proofs*. arXiv:2505.09095 (2025). http://arxiv.org/abs/2505.09095 (accessed 2026-02-15).

[16] NIST. *Lennard-Jones Fluid Reference Calculations: Cuboid Cell*. https://www.nist.gov/mml/csd/chemical-informatics-group/lennard-jones-fluid-reference-calculations-cuboid-cell (accessed 2026-02-15).

[17] Kwiatkowski, M. & Stark, I. *The Continuous π-Calculus: A Process Algebra for Biochemical Modelling*. In *Computational Methods in Systems Biology* (LNCS 5307), Springer (2008). DOI: 10.1007/978-3-540-88562-7_11. (Metadata: https://www.research.ed.ac.uk/en/publications/the-continuous-%CF%80-calculus-a-process-algebra-for-biochemical-model, accessed 2026-02-15).

[18] continuouspi. *lean-cpi: machine formalisation of the Continuous π-calculus in Lean*. https://github.com/continuouspi/lean-cpi (accessed 2026-02-15).

[19] Project archive. *A Formalisation of Biochemical …(undergraduate project report; mechanised continuous-π calculus in Lean)*. https://project-archive.inf.ed.ac.uk/ug4/20201778/ug4_proj.pdf (accessed 2026-02-15).

[20] AIChE Proceedings. *Logical Proofs about the Genetic Code in Lean 4*. (2024). https://proceedings.aiche.org/conferences/aiche-annual-meeting/2024/proceeding/paper/logical-proofs-about-genetic-code-lean-4 (accessed 2026-02-15).

[21] Holliday, W. H., Norman, C., & Pacuit, E. *Voting Theory in the Lean Theorem Prover*. (2021). https://escholarship.org/uc/item/2g73d7qv (accessed 2026-02-15).

[22] Pusceddu, D. & Bartoletti, M. *Formalizing Automated Market Makers in the Lean 4 Theorem Prover*. FMBC 2024 (OASIcs 118), 2024. DOI: 10.4230/OASIcs.FMBC.2024.5. https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.FMBC.2024.5 (accessed 2026-02-15).

[23] lecopivo. *SciLean: Scientific computing in Lean 4*. https://github.com/lecopivo/SciLean (accessed 2026-02-15).

[24] LeanCert project. *Verified numerical computation and bound certification for Lean 4*. https://docs.leancert.io/ (accessed 2026-02-15).

[25] ufmg-smite. *Lean-SMT: tactics for discharging Lean goals into SMT solvers*. https://github.com/ufmg-smite/lean-smt (accessed 2026-02-15).

[26] LeanDojo. *LeanDojo-v2: a comprehensive library for AI-assisted theorem proving in Lean 4*. https://leandojo.org/leandojo.html (accessed 2026-02-15).

[27] Poiroux, A. *LeanInteract: a Python interface for Lean 4 through the Lean REPL*. https://github.com/augustepoiroux/LeanInteract (accessed 2026-02-15).

[28] Murphy, L. et al. *Autoformalizing Euclidean Geometry*. ICML 2024. Code/benchmark: https://github.com/loganrjmurphy/LeanEuclid (accessed 2026-02-15).

[29] Gadgil, S. *LeanAide: AI-based tools for helping with Lean 4 (autoformalization-centric)*. https://github.com/siddhartha-gadgil/LeanAide (accessed 2026-02-15).

[30] Chojecki, P. *UnsolvedMath: dataset of open mathematical problems*. Hugging Face dataset page. https://huggingface.co/datasets/ulamai/unsolved_math_problems (accessed 2026-02-15).

[31] Chojecki, P. *Open Mathematical Problems as an AI Reasoning Benchmark*. ulam.ai (PDF, 2026-01-31). https://www.ulam.ai/research/open-math.pdf (accessed 2026-02-15).

[32] ulamai. *ulamai: agentic research/autoformalization stack (code repository)*. https://github.com/ulamai/ulamai (accessed 2026-02-15).

[33] Poiroux, A., Weiss, G., Kunčak, V., & Bosselut, A. *Improving Autoformalization using Type Checking*. arXiv:2406.07222 (2024). https://arxiv.org/abs/2406.07222.

[34] Azerbayev, Z., Piotrowski, B., Schoelkopf, H., Ayers, E. W., Radev, D., & Avigad, J. *ProofNet: Autoformalizing and Formally Proving Undergraduate-Level Mathematics*. arXiv:2302.12433 (2023). https://arxiv.org/abs/2302.12433.

[35] Qian, Y., Clune, J., Barrett, C. W., & Avigad, J. *Leanauto: An Interface between Lean 4 and Automated Theorem Provers*. arXiv:2505.14929 (2025). https://arxiv.org/abs/2505.14929.

[36] Holliday, W. H., Norman, C., Pacuit, E., & Zahedian, S. *Impossibility theorems involving weakenings of expansion consistency and resoluteness in voting*. arXiv:2208.06907 (2022; revised 2023). https://arxiv.org/abs/2208.06907.

[37] Huang, X. et al. *Machine Learning for Electronic Design Automation: A Survey*. arXiv:2102.03357 (2021). DOI: 10.48550/arXiv.2102.03357. https://arxiv.org/abs/2102.03357 (accessed 2026-02-15).

[38] Mirhoseini, A. et al. *A graph placement methodology for fast chip design*. *Nature* **594**, 207–212 (2021). DOI: 10.1038/s41586-021-03544-w. https://pubmed.ncbi.nlm.nih.gov/34108699/ (accessed 2026-02-15).

[39] Yue, S., Songhori, E. M., Jiang, J. W., Boyd, T., Goldie, A., Mirhoseini, A., & Guadarrama, S. *Scalability and Generalization of Circuit Training for Chip Floorplanning*. In *Proceedings of the 2022 International Symposium on Physical Design (ISPD '22)*, 65–70 (2022). DOI: 10.1145/3505170.3511478. https://dl.acm.org/doi/10.1145/3505170.3511478 (accessed 2026-02-15).

[40] Cheng, C.-K., Kahng, A. B., Kundu, S., Wang, Y., & Wang, Z. *Assessment of Reinforcement Learning for Macro Placement*. In *Proceedings of the 2023 International Symposium on Physical Design (ISPD '23)*, 158–166 (2023). DOI: 10.1145/3569052.3578926. https://arxiv.org/abs/2302.11014 (accessed 2026-02-15).

[41] Markov, I. L. *The False Dawn: Reevaluating Googles Reinforcement Learning for Chip Macro Placement*. arXiv:2306.09633 (2023). DOI: 10.48550/arXiv.2306.09633. https://arxiv.org/abs/2306.09633 (accessed 2026-02-15).

[42] Wang, Z. et al. *Benchmarking End-To-End Performance of AI-Based Chip Placement Algorithms*. arXiv:2407.15026 (2024). DOI: 10.48550/arXiv.2407.15026. https://arxiv.org/abs/2407.15026 (accessed 2026-02-15).

[43] Ajayi, T. et al. *INVITED: Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project*. In *The 56th Annual Design Automation Conference (DAC '19)*, June 2–6, 2019. DOI: 10.1145/3316781.3326334. https://par.nsf.gov/servlets/purl/10171025 (accessed 2026-02-15).

[44] Cheng, R. & Yan, J. *On Joint Learning for Solving Placement and Routing in Chip Design*. *NeurIPS 2021* (2021). https://papers.neurips.cc/paper_files/paper/2021/file/898aef0932f6aaecda27aba8e9903991-Paper.pdf (accessed 2026-02-15).

[45] Cheng, R. et al. *The Policy-gradient Placement and Generative Routing Neural Networks for Chip Design*. *NeurIPS 2022* (2022). https://papers.neurips.cc/paper_files/paper/2022/file/a8b8c1ad51df1b93d9e3d1fca75debbf-Paper-Conference.pdf (accessed 2026-02-15).

[46] Wang, Y. et al. *MCP4EDA: LLM-Powered Model Context Protocol RTL-to-GDSII Automation with Backend Aware Synthesis Optimization*. arXiv:2507.19570 (2025). DOI: 10.48550/arXiv.2507.19570. https://arxiv.org/abs/2507.19570 (accessed 2026-02-15).

[47] opencompl. *sail-riscv-lean: RISC-V ISA Semantics for Lean*. GitHub repository. https://github.com/opencompl/sail-riscv-lean (accessed 2026-02-15).

[48] Cipollina, M., Karatarakis, M., & Wiedijk, F. *Formalized Hopfield Networks and Boltzmann Machines*. arXiv:2512.07766 (2025). DOI: 10.48550/arXiv.2512.07766. https://arxiv.org/abs/2512.07766 (accessed 2026-02-17).

[49] Sonoda, S., Kasaura, K., Mizuno, Y., Tsukamoto, K., & Onda, N. *Lean Formalization of Generalization Error Bound by Rademacher Complexity*. arXiv:2503.19605 (2025). DOI: 10.48550/arXiv.2503.19605. https://arxiv.org/abs/2503.19605 (accessed 2026-02-17).

[50] Zhang, Y., Lee, J. D., & Liu, F. *Statistical Learning Theory in Lean 4: Empirical Processes from Scratch*. arXiv:2602.02285 (2026). DOI: 10.48550/arXiv.2602.02285. https://arxiv.org/abs/2602.02285 (accessed 2026-02-17).

[51] Andric, S. *BlockCert: Certified Blockwise Extraction of Transformer Mechanisms*. arXiv:2511.17645 (2025). DOI: 10.48550/arXiv.2511.17645. https://arxiv.org/abs/2511.17645 (accessed 2026-02-17).

[52] Nipkow, T., Paulson, L. C., & Wenzel, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science, vol. 2283. Springer (2002). DOI: 10.1007/3-540-45949-9. https://link.springer.com/book/10.1007/3-540-45949-9 (accessed 2026-02-17).

[53] Blanchette, J. C., Böhme, S., & Paulson, L. C. *Extending Sledgehammer with SMT Solvers*. *Journal of Automated Reasoning* **51**, 109–128 (2013). DOI: 10.1007/s10817-013-9278-5. https://link.springer.com/article/10.1007/s10817-013-9278-5 (accessed 2026-02-17).

[54] Immler, F. & Hölzl, J. *Numerical Analysis of Ordinary Differential Equations in Isabelle/HOL*. In *Interactive Theorem Proving (ITP 2012)*, LNCS 7406, 377–392. Springer (2012). DOI: 10.1007/978-3-642-32347-8_26. https://link.springer.com/chapter/10.1007/978-3-642-32347-8_26 (accessed 2026-02-17).

[55] Melquiond, G. *Floating-point arithmetic in the Coq system*. *Information and Computation* **216**, 14–23 (2012). DOI: 10.1016/j.ic.2011.09.005. https://www.sciencedirect.com/science/article/pii/S0890540112000739 (accessed 2026-02-17).

[56] Harrison, J. *HOL Light: An Overview*. In *Theorem Proving in Higher Order Logics (TPHOLs 2009)*, LNCS 5674, 60–66. Springer (2009). DOI: 10.1007/978-3-642-03359-9_4. https://link.springer.com/chapter/10.1007/978-3-642-03359-9_4 (accessed 2026-02-17).

Table 1: Selected Lean artifacts relevant to scientific, socio-economic, and adjacent engineering formalization (representative, not exhaustive).

Domain	Artifact	Lean	Formalized contribution	Why it matters for science
Physics	PhysLean [8, 9]	4	Topic-organized library (Maxwell, QHO, SR, ensembles, tight-binding, Higgs, Wick).	Signals *library-building* for physics rather than one-off proofs.
Physics	Index notation [10]	4	Verified tensor index notation; category-theoretic backend; physicist-friendly bridge.	Encodes implicit summation/index hygiene as obligations, reducing bookkeeping errors.
Physics/Chem	Dimensional analysis [12]	4	Dimensions as an Abelian group; SI units/constants; Buckingham Π theorem; LJ example.	Cross-domain correctness layer: unit/consistency proofs become routine.
Chemistry	Chemical-physics derivations [13, 14]	3/4	Langmuir and BET adsorption derivations; explicit premises; reusable thermodynamics/kinematics structures.	Mechanizes "derivation hygiene"; exposes hidden constraints (e.g., denominators).
Chemistry	LeanLJ energy kernel [15, 16]	4	Verified Lennard–Jones energy computation under periodic boundaries; Real/Float bridge; IO boundary; matches NIST benchmarks.	Demonstrates *verified scientific computation* with external conformance targets.
Biology	Continuous π-calculus in Lean [18, 19, 17]	3	Mechanized syntax/semantics for a biochemical process language; computable ODE generation in examples.	Semantics-first biology: makes "what the model means" explicit and checkable.
Biology	Genetic code properties [20]	4	Genetic code as a function (codons $\rightarrow$ amino acids); proofs of redundancy and non-overlap.	Pattern for "code-like biology": prove invariants of mappings and transformations.
Economics	Voting theory framework [21]	3/4	Variable-election encodings; dependent types for adding/removing voters/candidates; verification of method properties.	Mechanizes quantifier-heavy axioms where informal proofs are brittle.
Economics/DeFi	AMMs in Lean 4 [22]	4	Formal constant-product AMM model; mechanized economic properties including arbitrage.	Auditable market-mechanism semantics; useful for protocol verification and policy analysis.
Hardware	Sail $\rightarrow$ Lean RISC-V ISA semantics [47]	4	Translation of the official Sail RISC-V ISA specification into Lean; large, bitvector-heavy formal semantics that type-check as a library artifact.	Provides a certified hardware–software interface layer; a natural anchor for linking AI-driven EDA outputs and hardware verification claims to small-kernel checking.

Track	Task	Input → Output	Success criterion
SciAF-Stmt	Statement autoformalization	NL claim (+ context) → Lean theorem statement	Elaborates in the declared environment; passes domain validity predicates (DVPs).
SciAF-Assump	Assumption completion	NL claim + failed Lean attempt → repaired statement (added hypotheses)	Elaborates; admits a proof in Lean (kernel-checked when possible); hypotheses are *minimized* by ablation tests.
SciAF-Proof	Proof synthesis	Lean statement → proof term / tactic script	Goal closed with kernel checking; trust boundary recorded if external solvers are used without reconstruction.
SciAF-Exec	Spec → executable kernel	Lean spec + reference IO/benchmark → Lean (or Lean+FFI) executable	Executable matches reference benchmarks within declared tolerance; core invariants proved; IO isolated as a boundary [15].
SciAF-DVP (science-only)	Domain validity enforcement	Candidate Lean statement/proof → acceptance / rejection	Candidate must satisfy DVPs (units, tensor shapes, well-formed states); otherwise rejected even if it elaborates.
SciAF-Trust (science-only)	Trust accounting	Completed artifact → trust label + budget	Report which components are kernel-checked vs. externally trusted (ATP/SMT/numerics); prefer proof reconstruction [25, 35, 24].

Table 2: SciAF task suite. SciAF extends standard autoformalization with domain validity predicates (DVPs) and explicit trust accounting, reflecting scientific needs (units, shapes, state invariants, numeric boundaries).

Class	Typical hidden assumption	How Lean surfaces it	Representative domains
Non-vanishing	Denominators $\neq 0$; invertibility of matrices/operators	Goals of the form $a \neq 0$ after `field_simp`; invertibility obligations	Chem derivations (Langmuir/BET) [13]; econ formulas with ratios [22]
Positivity / order	Positivity of rates, probabilities, concentrations, reserves; monotonicity conditions	`linarith` goals; order constraints; side goals $0 < x$ for inequalities	Chem/biology rates; AMM reserves; econ constraints [22]
Regularity	Differentiability / integrability / measurability assumptions	`simp` leaves obligations `Measurable`, `Integrable`, `HasDerivAt`, etc.	Physics/chem calculus; stochastic econ/biology models (future scale)
Well-formedness	Units/dimensions consistent; tensor indices valid; state invariants hold	DVP failures (dimension check; index-range checks; invalid state)	Units (physics/chem) [12]; tensors [10]; AMM states [22]
Semantic scoping	Binder/name scoping; substitution conditions; structural congruence premises	Scope-related typing failures; obligations about context membership or substitution lemmas	Biochemical process calculi [18]
Approximation boundary	Floating-point / discretization error bounds; tolerance regimes	Explicit refinement goals relating `Real`-spec to `Float`-exec, or certificate checks	Verified computation (LeanLJ) [15]; certified bounds [24]

Table 3: Hidden-assumption taxonomy (pilot template). Lean exposes "implicit" scientific premises as explicit proof obligations. The ledger converts these obligations into supervised data for SciAF-Assump and into actionable modeling guidance (which premises should be packaged as reusable structures).