Proving Your Way to Cooperation: Formalizing Proof-Based Open Source Game Theory in Lean

Colomban Duclaux [*1 2] **Riccardo Formenti** [*1] **Pepijn Cobben** [1 2] **Bernhard Schölkopf** [1 3 4] **Zhijing Jin** [2 3 5]

Abstract

Open Source Game Theory studies strategic interaction between agents with full mutual transparency, and offers a promising route to robust cooperation in multi-agent systems. Its proof-based branch, in which agents condition on bounded proof search over each other's source code, is theoretically rich, but lacks any mechanized framework, limiting both its empirical study and its integration into practical systems. We address this gap with the first Lean 4 formalization of proof-based Open Source Game Theory, comprising nine open-source programs together with manually written, machine-checked proofs of the outcome theorems that govern their pairwise play. On top of this verified library, we build an agentic pipeline that turns natural-language strategy descriptions into Lean-verified outcome theorems. This pipeline autonomously reproves 40 of the 45 outcome theorems in the resulting bot-pair matrix. As a first application, we use the verified outcomes to enumerate the Nash equilibria of the induced Prisoner's Dilemma program meta-game and find a broad spectrum of cooperative equilibria absent from the underlying game[1].

[1]ETH Zürich, Switzerland [2]EuroSafeAI, Zürich, Switzerland [3]Max Planck Institute for Intelligent Systems Tübingen, Germany [4]ELLIS Institute Tübingen, Germany [5]Jinesis Lab, University of Toronto & Vector Institute, Canada. Correspondence to: Colomban Duclaux <cduclaux@ethz.ch>, Riccardo Formenti <rformenti@ethz.ch>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026. Copyright 2026 by the author(s).

[1]Code accompanying this paper, pinned to the version described in the paper: https://github.com/ColombanD/open-source-game-theory/tree/ai4math-icml-camera-ready. The actively developed version lives on `main`.

1. Introduction

Sustaining cooperation is an enduring challenge of human society. The incentives against it are often strong: producing arms in an arms race (Schelling, 1960), increasing emissions under a shared climate, or overfishing a common lake (Hardin, 1968; Ostrom, 1990) all instantiate dilemmas in which unilateral noncooperation pays off. The difficulty is that many socially valuable outcomes are individually fragile and each agent may have a private incentive to free-ride, defect, or overexploit, even when mutual restraint would leave the group strictly better off.

These dynamics (and their solutions) are usually expressed in the language of game theory, the mathematical theory of strategic decision-making (Osborne & Rubinstein, 1994). Classically, however, game-theoretic models represent players only through their actions and beliefs, rather than as inspectable decision procedures; whatever an agent does not know about its opponents' payoffs, feasible strategies, or information is then folded into the apparatus of incomplete information (Harsanyi, 1967; 1968a;b). In many strategically important settings this black-box assumption imposes more opacity than reality warrants: the laws and procedures of institutions are typically public, individuals can disclose

```
def DupocBot(k, opp):
    if proof_search(k, opp,
            "opp(DupocBot(k)) == C"):
        return C
    else:
        return D
```

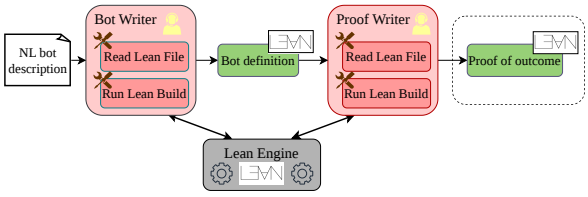

Figure 1. (Top) `DupocBot`, a canonical Open Source Game Theory program: cooperate iff bounded proof search finds the opponent will cooperate (Critch et al., 2022). (Bottom) Our agentic pipeline, translating natural-language program descriptions into machine-checked outcome theorems.

their principles and their past behavior, and the internal states of AI models are increasingly readable (Bereska & Gavves, 2024).

In this work, we study the limiting case of such transparency, Open Source Game Theory (OSGT), in which each player has complete knowledge of its opponents' source code. Interestingly, OSGT offers hope that cooperation may be substantially easier to achieve (and, conversely, collusion harder to prevent). By conditioning its action on a verifiable property of the opponent's program, an agent can construct strategies that are stable: they cooperate with sufficiently cooperative opponents while remaining robust against exploitation by adversarial ones (Tennenholtz, 2004; Bárász et al., 2014; Critch et al., 2022). Figure 1 (Top) gives the canonical example, a program that defects unless it can verify, via bounded proof search, that its opponent cooperates against it.

OSGT is nonetheless theoretically demanding and computationally intensive, which has limited both its growth and the concrete insights it has produced. One especially promising direction is proof-based OSGT (Bárász et al., 2014; Critch et al., 2022), in which formal logic is used to reason about strategic outcomes. While prior efforts have mechanized classical game theory (MixedMatched, 2023), to our knowledge, none have targeted the proof-based OSGT setting specifically. Such a framework is crucial in that it would let us test the viability of the theory in concrete settings, integrate it into practical AI systems, and analyze its equilibria far more systematically than hand computation allows.

Our contributions are as follows.

- **Lean formalization of proof-based OSGT.** We introduce a Lean 4 library that formalizes the basic setting of (proof-based) OSGT (§3.1–3.2, Appendix A). We populate it with a concrete suite of open-source agents and hand-written machine-checked outcome proofs for the bot-pair matrix, mechanizing in particular the provability-conditioned constructions of Critch et al. (2022).
- **Agentic pipeline.** On top of this verified library, we build an agentic pipeline (Figure 1, Bottom) that turns natural-language strategy descriptions into Lean programs and then attempts to prove their pairwise outcome theorems (§3.3, Appendix B). The pipeline follows a retrieve-propose-check architecture with agents reading relevant Lean files, proposing candidate bot definitions or proofs, running the Lean kernel, and iterating on compiler feedback. In our evaluation, the proof agent autonomously proves 40 of the 45 unordered outcome theorems in the hand-written bot library.
- **Equilibrium analysis of the program meta-game.** As a first application, we use the verified outcome matrix to enumerate the Nash equilibria of the restricted program meta-game induced by the bot library (§4.2, Appendix C). We find that the program meta-game admits a full spectrum of cooperative equilibria absent from the underlying Prisoner's Dilemma, including full-cooperation equilibria attaining the efficient payoff. We further identify `DupocBot` as the unique element of the library whose presence is necessary for full mutual cooperation to arise.

2. Theory

2.1. Open Source Game Theory

A *game* is, informally, a strategic environment in which two or more *players* interact, form beliefs about each other, reason, and ultimately play an *action* from a list of available actions, receiving a *payoff* that depends on the choices of the co-players. Formally, a game G in normal form is a tuple $G = (I, (A_i, u_i)_{i \in I})$, where I is a finite set of players, A_i is the finite set of actions available to player i, and $u_i \colon \prod_{j \in I} A_j \to \mathbb{R}$ is the payoff (utility) function of player i. We allow players to randomize: a *mixed action* for player i is a probability distribution over A_i, i.e. an element of the simplex $\Delta(A_i)$, and we extend u_i to $\prod_{j \in I} \Delta(A_j)$ by expectation in the usual way. As is customary, we denote other players' actions by $a_{-i} \in A_{-i} := \prod_{j \neq i} A_j$.

The guiding question of game theory is which action profiles can be regarded as the outcome of play by *rational* players, that is, which actions are argumentatively justified by tenets of rationality. The classical answer is the *Nash Equilibrium* (NE) (Nash, 1950; 1951): an action profile $(\alpha_i)_{i \in I} \in \prod_{j \in I} \Delta(A_j)$ such that no player can strictly increase their payoff by unilaterally deviating to a different (mixed) action. Although Nash equilibria are individually rational in this sense, they need not be collectively so: there might exist a more efficient profile that every player prefers[2]; the canonical example is mutual defection in the Prisoner's Dilemma (PD), which is the unique NE and yet leaves both players strictly worse off than they would be under mutual cooperation.

The PD inefficiency above hinges on a form of strategic opacity: each player commits to an action with no verifiable knowledge of what their opponent will do. Suppose instead that strategies are not actions but computer *programs*, and that players submit these programs which are mutually readable before any of them is executed. A player's program may then condition its action on a formal inspection of the opponents' programs, and conditional commitments of the form "cooperate if and only if I can verify that you will cooperate" become well-defined objects of strategic choice.

[2]Here, we mean efficient in the sense of Pareto: an action profile a is Pareto-dominated by another a' if for all players i, $u_i(a') \geq u_i(a)$, with strict inequality for at least one player; a profile is then Pareto efficient if no profile Pareto-dominates it.

This setting was introduced by Tennenholtz (2004), who observed that the natural solution concept to this new setting is simply the Nash equilibrium of the meta-game in which each player's strategy set is the space of admissible programs; he called such equilibrium a *Program Equilibrium* (PE). Critch et al. (2022) later named the broader area *Open-Source Game Theory* (OSGT) and argued for its relevance to interacting software systems and to human institutions; the provability-logic line initiated by Bárász et al. (2014) is particularly relevant to the present paper. The enlarged strategy space yields strictly more equilibria than the NE on the underlying game, with mutual cooperation in the one-shot PD being the simplest illustration. We return to it in §4.2; see also Appendix C.

Fix an underlying game G (e.g. the Prisoner's Dilemma). We now make precise the meta-game in which each player, rather than committing directly to an action, submits a *program* that takes the other players' programs as input and outputs an action of G. For each player i, let Σ_i denote a set of *admissible* programs for player i, and write $\Sigma := \prod_{i \in I} \Sigma_i$ the set of admissible program profiles. We deliberately leave admissibility under-specified at this stage, with the only blanket requirement being that every admissible program $\pi_i \in \Sigma_i$ halt on every input $\pi_{-i} \in \Sigma_{-i}$ and return an element of A_i. A concrete choice of Σ, the one used in our formal development, is given in §3.1 and Appendix A, where we specify admissibility inside a Lean 4 library.

A profile $\pi = (\pi_i)_{i \in I} \in \Sigma$ of admissible programs induces an action profile in $\prod_i \Delta(A_i)$ by simultaneous execution: each π_i is run on π_{-i}, and the resulting actions are played in G. Formally, define

$$\mathrm{outcome}\colon \Sigma \longrightarrow \prod_{i \in I} A_i, \quad \mathrm{outcome}(\pi)_i := \pi_i(\pi_{-i}).$$

Composing with the payoffs of G yields the meta-game payoffs

$$U_i\colon \Sigma \longrightarrow \mathbb{R}, \qquad U_i(\pi) \;:=\; u_i\big(\mathrm{outcome}(\pi)\big).$$

The tuple $G^{\mathrm{prog}} := \big(I, (\Sigma_i, U_i)_{i \in I}\big)$ constitutes a normal-form game with possibly countably infinite pure-strategy spaces, which we call the *program meta-game* associated with G. A mixed strategy for player i in G^{prog} is a probability distribution $\mu_i \in \Delta(\Sigma_i)$ over admissible programs; payoffs extend to $\prod_i \Delta(\Sigma_i)$ by expectation.

Definition 2.1 (Program Equilibrium)**.** A profile $\pi^* \in \Sigma$ is a (pure) *Program Equilibrium* (PE) of G if it is a pure-strategy Nash equilibrium of G^{prog}: for every $i \in I$ and every $\pi_i' \in \Sigma_i$,

$$U_i(\pi_i^*, \pi_{-i}^*) \;\geq\; U_i(\pi_i', \pi_{-i}^*).$$

More generally, a profile $\mu^* \in \prod_i \Delta(\Sigma_i)$ is a *mixed Program Equilibrium* if it is a (mixed) Nash equilibrium of G^{prog}.

Whenever the underlying game G admits a pure-strategy Nash equilibrium $(a_i^*)_{i \in I} \in \prod_{i \in I} A_i$, the meta-game G^{prog} admits a corresponding pure-strategy PE: take π_i^* to be the constant program that ignores its input and returns a_i^* (assuming, as we shall throughout, that such constant programs lie in Σ_i). Any unilateral deviation $\pi_i' \in \Sigma_i$ induces, against the constant π_{-i}^*, the action profile $(\pi_i'(\pi_{-i}^*), a_{-i}^*)$, whose payoff for player i is at most $u_i(a_i^*, a_{-i}^*)$ by the NE property of a^*, which is precisely the payoff for player i. In particular, every game with a pure NE admits a pure PE.

The same lifting strategy yields a mixed-strategy existence result for all finite games by randomizing over constant programs, under the non-triviality assumption that for each $i \in I$ and each $a \in A_i$, the constant program that ignores its input and outputs a is admissible. By abuse of notation we denote this program by $a \in \Sigma_i$:

Theorem 2.2 (Existence of mixed PE)**.** *Let G be a finite normal-form game, and suppose that for each $i \in I$ the set of admissible programs Σ_i is non-trivial. Then the meta-game G^{prog} admits a mixed-strategy PE.*

Proof. By Nash's theorem (Nash, 1950) applied to the finite game G, there exists a mixed NE $\alpha^* = (\alpha_i^*)_{i \in I}$. Define $\mu^* = (\mu_i^*)_{i \in I} \in \prod_{i \in I} \Delta(\Sigma_i)$ by

$$\mu_i^*(a) \;:=\; \alpha_i^*(a) \qquad (i \in I,\ a \in A_i),$$

with all remaining mass on Σ_i set to zero. We show μ^* is a mixed PE of G^{prog}. Fix $i \in I$ and a deviation $\mu_i' \in \Delta(\Sigma_i)$. Since every program in the support of μ_{-i}^* is constant, let $\alpha_i' \in \Delta(A_i)$ denote the marginal action distribution induced by μ_i' against μ_{-i}^*,

$$\alpha_i'(a) \;:=\; \sum_{\pi_i' \in \Sigma_i} \mu_i'(\pi_i') \cdot \mathbf{1}_{\left[\pi_i'(a_{-i}^*) = a\right]}.$$

Then

$$\begin{aligned}\mathbb{E}_{\pi \sim (\mu_i', \mu_{-i}^*)}\big[U_i(\pi)\big] \;=\; u_i\big(\alpha_i', \alpha_{-i}^*\big) \;&\leq\; u_i\big(\alpha_i^*, \alpha_{-i}^*\big) \\ &=\; \mathbb{E}_{\pi \sim \mu^*}\big[U_i(\pi)\big],\end{aligned}$$

where the inequality is the NE property of α^* in G. □

The interest of G^{prog} over G comes entirely from the "open reading" feature of the meta-game: π_i has full syntactic access to π_{-i}. Since source codes are finite strings, anything computable about them is in principle available to π_i when it decides its action. In particular, π_i may:

1. **Simulate** π_j on chosen inputs. This is the direction pursued by the simulation and recursion based program-equilibrium line of Oesterheld (2019), Kovarík et al. (2023; 2024), and Cooper et al. (2025).
2. **Compute syntactic predicates** of π_j, for instance, the predicate $\pi_j = \pi_i$ exploited by Tennenholtz's CliqueBot. (Tennenholtz, 2004).

3. **Search for formal proofs** about π_j: for a fixed proof system T (e.g. Peano Arithmetic), enumerate proofs in T up to a length bound k and check whether any of them proves a chosen statement about π_j. This is the provability-logic line initiated by Bárász et al. (2014) and developed by Critch et al. (2022).

Options (1) and (3) are closely related, a successful simulation is, in effect, a concrete computational proof of the simulated bot's behavior, but they differ sharply in what they make legible to the *other* bot. A simulation result is opaque outside the simulator; a formal proof, by contrast, is a finite string in a fixed proof system that any other bot can independently re-verify. This is the feature that lets provability-conditioned bots achieve mutually visible common ground without an unbounded mutual-simulation regress (Critch et al., 2022).

The rest of the paper focuses on (3). Concretely, this makes available programs whose decision rule is of the form

> "play $a \in A_i$ iff T proves, in at most k symbols, that $\varphi(\pi_{-i})$,"

for a chosen arithmetical formula φ encoding a property of the opponents' programs (typically: "the opponent cooperates against me"). The strategic analysis of G^{prog} thus reduces itself, for programs of this form, to questions about the proof-theoretic structure of T, which will be made precise in §3.1. As we shall see, this yields equilibrium phenomena (mutual cooperation in the one-shot PD, self-fulfilling defection from cooperative-looking criteria, and so on) that have no analogue in the Nash analysis of G.

2.2. Cooperation in the Prisoner's Dilemma

The Prisoner's Dilemma (PD) will serve as the canonical illustration of the open-source phenomena described in §2.1. Two players simultaneously choose between cooperation (C) and defection (D), with symmetric payoffs

$$\begin{array}{c|cc} & C & D \\ \hline C & (R,R) & (S,T) \\ D & (T,S) & (P,P) \end{array} \quad , T > R > P > S.$$

The action D strictly dominates C for each player, so (D, D) is the unique Nash equilibrium of G, despite being Pareto-dominated by (C, C). Lifting to the meta-game G_{prog} preserves this equilibrium: the constant-D profile is a pure PE by the same argument as Theorem 2.2. The relevant question is whether the enlarged strategy space Σ admits *new* equilibria in which the players cooperate.

The first candidate is the program that ignores its input and unconditionally cooperates:

```
def CooperateBot(opp):
    return C
```

`CooperateBot` is not part of any PE in which both players use it: against (`CooperateBot`, `CooperateBot`) either player profits by deviating to the constant-D program `DefectBot` and securing $T > R$. To support cooperation in PE, a cooperative program must condition its action on a verifiable property of its opponent. Tennenholtz (2004) achieves this with the syntactic criterion "cooperate iff the opponent's source code equals mine," which yields a PE between two identical copies but breaks under any rewriting of either bot. We seek a strictly more robust criterion, in which cooperative intent is recognized in opponents whose source codes are allowed to differ syntactically.

The provability-based design of Bárász et al. (2014) and Critch et al. (2022) provides such a criterion. Fix a formal theory T and assume access to a length-bounded proof search,

```
def proof_search(k, opp, hypothesis):
    # Returns True iff T has a proof of
    # 'hypothesis' about 'opp' of length
    # at most k.
```

We require throughout that `proof_search` be *sound*: if `proof_search`$(k, \texttt{opp}, \varphi)$ returns `True`, then φ is provable in T. Equipped with such an oracle, the canonical provability-based program is `DupocBot`(k), "Defect Unless Proof of Cooperation",

```
def DupocBot(k)(opp):
   if proof_search(k, opp, "opp(DupocBot(
    k)) == C"):
        return C
   else:
        return D
```

`DupocBot`(k) cooperates only when it has produced, in at most k symbols, a proof in T that its opponent cooperates against it; otherwise it falls back to defection.

Soundness of `proof_search` immediately gives unexploitability: `DupocBot`(k) never receives the sucker's payoff S, because any cooperation on its part is witnessed by an actual proof of opponent cooperation (Critch et al., 2022, Prop. 3.2). A fascinating and counterintuitive result of Critch et al. (2022) concerns the self-play behavior of `DupocBot`(k) for k sufficiently large: against itself, `DupocBot`(k) *cooperates* (Critch et al., 2022, Thm. 3.7); in fact, the Parametric Bounded Löb Theorem (PBLT; Critch, 2019) yields a short proof of `DupocBot`(k)(`DupocBot`$(k)) = C$, which each copy then locates within its budget k and acts upon. This is the central positive phenomenon of the open-source set-

ting: a program whose default action is to *defect* reaches the cooperative outcome (C, C) in self-play. The profile $(\texttt{DupocBot}(k), \texttt{DupocBot}(k))$ is moreover a PE of G^{prog} realizing the Pareto-optimal payoff (R, R) as the only profitable unilateral deviation would be one that secures the temptation payoff T against $\texttt{DupocBot}(k)$, and unexploitability rules this out. The meta-game G^{prog} thus admits a Nash equilibrium attaining the Pareto-optimal payoff that the underlying game G structurally rules out. We return to this and further such phenomena in §4.2.

The mirror-image construction $\texttt{CupodBot}(k)$ ("Cooperate Unless Proof of Defection"), which cooperates by default and defects only on proof of opponent defection, exhibits the opposite phenomenon: despite its cooperative-looking default, PBLT yields mutual *defection* in self-play (Critch et al., 2022, Thm. 3.4). The labels "cooperative-looking" and "defection-looking" thus swap roles once self-reference and bounded proof search enter the picture, and the equilibrium analysis becomes correspondingly delicate. This is the principal motivation for the formal treatment in §3.1: claims of the form "$\texttt{outcome}(A, B) = (a_1, a_2)$" depend on subtle interactions between source-code self-reference, the length bound k, and the proof-theoretic profile of T, precisely the setting in which a proof assistant, such as Lean, becomes useful.

3. Lean Formalization and Automation

3.1. The Lean Engine

We formalize the program-equilibrium framework of Bárász et al. (2014) and Critch et al. (2022) on PD as a Lean 4 library, `PrisonersDilemma`. Despite prior suggestions to use HOL (Nipkow et al., 2002) or Coq (Chlipala, 2013), we choose Lean 4 for two specific reasons: (i) Programs are written as terms of an inductive type and the outcome semantics is an executable function, so many outcome theorems close by reduction alone; and (ii) current LLM proof agents are substantially stronger in Lean than in Coq or Isabelle, which is essential for the automation pipeline of §3. Full definitions appear in Appendix A.

Programs. Strategies are terms of an inductive type `Prog` with seven constructors covering the constant action `.const`, opponent simulation `.sim`, conditional branching on a simulated action `.ite`, bot references `.bot`, the two free variables `.self` and `.opp` (placeholders for "my source code" and "the opponent's source code"), and proof-search conditioned execution `.search`. Together, these constructors give a concrete inductive realization of the set Σ of admissible program profiles from §2.1. The semantic content of self-reference (a program reasoning about its own code) is handled by a syntactic substitution `subst : Prog → Prog → Prog → Prog` applied at evaluation time; see Appendix A.1 for the exact definition. This separates the static type-theoretic structure of `Prog` from the dynamic act of "running A against B with each side seeing the other's code."

Evaluation under fuel. The evaluator `eval : Nat → (Prog × Prog × Prog) → Action` formalizes what it means for a matchup to "output" an action. It is structurally recursive on a `Nat` fuel parameter. Because OSGT programs can simulate opponents that simulate them back, no a priori bound on evaluation depth exists; the fuel parameter is the device that recovers structural recursion. Moreover, the evaluator is `noncomputable`: its `.search` guard consults the proof oracle below, whose cooperative fixpoint outcomes enter through axioms rather than exhibited proof terms. We characterize this precisely in §A.2 and explain why it is a limitation of our axiomatic route rather than of bounded provability itself. The library's outcome theorems are then of the form

$$\forall n,\ \text{outcome}\,(n+k)\,(\text{Bot}_A, \text{Bot}_B) = \texttt{some}(a_1, a_2)$$

for a concrete offset $k \in \mathbb{N}$: once fuel exceeds k, the result stabilizes at the action pair (a_1, a_2).

Formulas and the proof oracle. As stated in §2.1, the subtle point in proof-based OSGT is in its proof search function, which in our library corresponds to the oracle `proofSearch : ℕ → Formula → Bool`. Defining this oracle requires two ingredients: an encoding of the hypotheses it reasons about (e.g. "you play D against me," "this implication is provable in $\leq k$ steps"), and a specification of what it means to search for a proof. We address them in turn.

Readers familiar with Lean might be tempted to encode hypotheses as `Prop`, but this would run into the same positivity problems that motivated the `Prog` encoding. We therefore model them not as Lean propositions but as values of a separate inductive type `Formula`. Constructors include the atomic predicate `.plays` $p\,q\,a$ ("program p plays action a against opponent q"), the bounded provability operator `.box` $k\,\varphi$ ("φ is derivable in $\leq k$ proof steps"), and the usual logical connectives. A separate `interp : Formula → Prop` recovers the propositional reading by mapping each constructor to its intended Lean proposition (e.g. `.plays` to the corresponding evaluation statement, connectives to their Lean counterparts), and `Formula.subst` resolves `.self` / `.opp` references inside formulas just as it does for programs.

With `Formula` in hand, the oracle's intended behavior is straightforward: `proofSearch` $k\,\varphi$ returns `true` when formula φ admits a proof of size $\leq k$ in our ambient theory T, and `.search` $k\,\varphi\,p\,q$ then runs p or q accordingly. We thus need to say what T is, or at least what we assume about it. There are two ways to do this. One can *internalize* T,

fix a concrete theory, encode its syntax and derivations in Lean, implement `proofSearch` as a bounded enumerator over T-derivations, and prove the required properties as theorems. Or one can *axiomatize* the interface: declare `proofSearch` opaque and postulate only the properties the library actually uses. We follow Critch et al. (2022) and take the axiomatic route, and we do so as a *deliberate scoping decision* rather than a shortcut, because it is orthogonal to this paper's contribution: a mechanized OSGT library with an associated bot collection and an LLM proof-automation pipeline. Additionally, axiomatizing the oracle interface keeps every OSGT result *parametric in* T: any theory satisfying A1–A8 induces a valid model, exactly mirroring the theory-agnostic stance of Critch et al. (2022).

Crucially, these axioms are not free-floating postulates that could silently introduce inconsistency. Each is a property that the internalize route would establish as a *theorem* for any standard bounded proof system, such as soundness and completeness of bounded proof search, code-reading, hypothetical syllogism, and the bounded Löb fixpoint of Critch et al. (2022), which is itself a proved theorem of arithmetic rather than a new assumption. We treat them as an interface specification, not as new mathematical content, and Appendix A.3 justifies each one by exhibiting its intended proof or citing the result it abstracts. Because the statements are concrete and the bot definitions are explicit Lean terms, every outcome theorem still has the property that *type-checking is correctness* relative to this fixed, audited interface.

The eight axioms (A1–A8, full statements in Appendix A.3) break into three groups. A1–A4 fix `proofSearch` as a bounded-provability decision procedure: soundness (A1), completeness for atomic `.plays` facts (A2), exact budget semantics ($\texttt{proofSearch}, k, \varphi = \texttt{true}$ iff some witness of size $\leq k$ proves φ, A3), and witness transport across formula families (A4). A5–A7 grant T the proof-theoretic moves the library needs without modeling its internals: code-reading for `.search` (A5) and `.sim` (A6) bodies, and hypothetical syllogism (A7). A8 is the Parametric Bounded Löb Theorem of Critch et al. (2022), the technical engine behind `CupodBot` and `DupocBot`. No further assumptions about T are introduced anywhere in the library.

3.2. Building the Bot and Proof Library

With the Lean framework in place, we populate it with a concrete library of OSGT programs (named with the suffix `Bot`) and their outcome proofs. This step matters for several reasons:

- it validates the coherence of the framework;
- it mechanizes the constructions of Critch et al. (2022) that motivated this work (notably `CupodBot` and `DupocBot`);
- it serves as few-shot context for the LLM proving agents (§3.3), as the ground-truth baseline for their evaluation (§4.1), and as the basis for the enumeration of Program Equilibria (§4.2).

We assemble nine bots, drawn from or inspired by Alex Mennen's transparent Prisoner's Dilemma tournament (Mennen, 2013) and Critch et al. (2022), together with their hand-written outcome proofs. The selection is small but covers the canonical archetypes that recur throughout the proof-based OSGT literature: constants, standard reactive baselines, and the provability-conditioned constructions of Critch et al. (2022). Table 3 shows all the proven outcomes. The bots span three complexity tiers: **Tier 0** (constant bots: `CooperateBot`, `DefectBot`); **Tier 1** (reactive bots that simulate the opponent on a probe input but do not use `.search`: `MirrorBot`, `TitForTatBot`, `DBot`, `EBot`, `OBot`); and **Tier 2** (`.search`-using bots: `CupodBot`, `DupocBot`). See Appendix A.4 for the exact definitions and Appendix A.5 for a sketch of a particular proof.

3.3. Agentic Proof Automation

With our Lean library fully built and filled with handwritten bots and proofs, we use it to implement an agentic pipeline that takes a natural-language strategy description, synthesizes a Lean 4 bot, and produces a machine-checked outcome theorem against any chosen opponent (see Figure 1). The pipeline consists of two independent LLM agents, a bot writer and a proof search agent, both following a retrieve-propose-check architecture: the agent reads relevant library files, proposes a candidate, runs the Lean kernel as a tool, and iterates on the compiler feedback until the artifact type-checks. All outcome theorems follow a fixed template, `outcome (n + k) X Y = some (.`a_1`, .`a_2`)`, with the proof agent free to pick the fuel offset k and action pair (a_1, a_2); this template is what makes evaluation tractable, since compilation against it equals correctness modulo the natural-language-to-Lean translation step. We expose the pipeline through a small FastAPI server with a minimal HTML/JavaScript front-end that runs the two agents asynchronously and gates each write to the library on human acceptance. The gate is binary: the human may only accept or reject the agent's output, never edit the generated bot source or proof. All evaluation results reported in §4 were obtained with this gate disabled, so no human intervention occurred inside the agent loop or between iterations.

Bot writer agent. Given a natural-language strategy description and a desired bot name, the bot writer produces a valid `.lean` file defining a `Prog` term. Its system prompt

embeds the full `Program.lean` and `Dynamics.lean` modules (the `Prog` grammar and its semantics), and the source of every hand-written bot in the library as few-shot examples. The agent has access to two tools: `run_lean_build`, which writes the candidate file to a temporary location inside the engine and invokes `lake env lean`; and `read_library_file`, which exposes any `.lean` file in the library. The loop terminates when `lake env lean` returns exit code 0 and a human has validated the lean script.

Proof search agent. Given a pair of bots, the proof-search agent autonomously discovers the outcome (action pair and fuel offset) and produces a Lean proof of the corresponding outcome theorem. It uses the same two tools as the bot writer and a system prompt of the same shape, extended with proof-specific instructions. The per-pair user prompt contains the target theorem statement, the source of both bots, and the retrieved few-shot context described below. When either target bot uses the `.search` constructor, `Axioms.lean` and `ProofSearch.lean` are additionally injected into the system prompt. See Appendix B.2 for the full prompt.

Retrieval. For each proof query, we retrieve up to four relevant theorem files from the existing library, ranked by structural match on bot names: files whose name matches one of the two target bots come first, then files that mention either bot in their body, then unrelated files as filler. We additionally pass the agent a compact summary of all already-proven outcome theorems involving either target bot. During evaluation, to prevent answer leakage, both the retrieval and the `read_library_file` tool filter out any file whose name matches a bot in the current query, and the known-theorems summary excludes the exact target pair.

Relation to prior work. Our pipeline instantiates the retrieve-propose-check loop established by Yang et al. (2023) with ReProver, and shown by COPRA (Thakur et al., 2023) to work in-context without fine-tuning. Domain-specialized provers, DeepSeek-Prover (Xin et al., 2024), TheoremLlama (Wang et al., 2024), Goedel-Prover-V2 (Lin et al., 2025), the reasoning-oriented Kimina-Prover (Wang et al., 2025), and editor integrations such as Lean Copilot (Song et al., 2025) primarily target general-purpose mathematics; to our knowledge none has been applied to mechanize OSGT or any other areas of game theory. A separate, complementary line of work evaluates LLMs as *players* in OSGT-style games (Sistla & Kleiman-Weiner, 2025) rather than as proof automators over a mechanized OSGT library, which is our focus here.

Table 1. Median iterations and pass rate stratified by bot complexity tier. Tier 0: constant; Tier 1: reactive (no `.search`); Tier 2: `.search`-using. Off-diagonal entries are unordered pairs.

Tier pair	Med. it.	Passed	Total
0×0	1	3	3
0×1	2	10	10
0×2	2.5	4	4
1×1	2	15	15
1×2	7	6	10
2×2	1	2	3
Total	**2**	**40**	**45**

4. Experimental Setup and Results

4.1. Proof Search Agent Evaluation

Setup. We evaluate the proof search agent on every unordered pair of bots in the hand-written library described in §3.2. Outcomes are symmetric in our template (`outcome` n A B and `outcome` n B A prove the same fact modulo argument order), so we run only the half-matrix: $\binom{9}{2} + 9 = 45$ pairs.

For each pair, the agent is given no claimed outcome and no fuel hint and it discovers both autonomously through the retrieve-propose-check loop. The retrieval stage is configured with the leak-prevention filters described in §3.3 (target bot files blocked at the tool layer; known-theorem summary excludes the exact target pair). We use Claude Opus 4.7 (Anthropic, 2026) as the proof agent with adaptive thinking enabled and a maximum of 20 tool-use iterations per pair. Sampling parameters are left at the API defaults (temperature 1.0). The agent has read-only access to the library. As noted in §3.3, a passing Lean build certifies the result.

Headline result. Out of 45 pairs, the agent produces a machine-checked Lean proof for **40 (88.9%)**. Tiers 0 and 1 are completely solved: every pair in which neither bot uses `.search` (32 pairs) is proved on the first attempt. All five remaining failures involve a Tier-2 bot. Across all 45 pairs the mean wall-clock time is 196 s (median 77 s), and the mean number of tool-use iterations is 3.4 (median 2). Table 1 stratifies the pass rate by tier pair (see Appendix B.1 for more details on results).

Output-budget sensitivity. We ran the experiment in two passes. The first pass used an output budget of 16,384 tokens per API call and produced 38/45 passes; all seven failures returned `stop_reason=max_tokens`, indicating that the agent was truncated mid-proof rather than reaching a dead end. We re-ran the failing pairs with the budget raised to 32,000 tokens (the per-call maximum supported by Opus 4.7 with extended thinking) and recovered two additional pairs, for a final pass rate of 40/45.

Failure analysis. The five remaining failures all in-

volve `CupodBot` or `DupocBot` paired with a Tier-1 or Tier-2 opponent. Proofs in this regime must chain completeness, witness transport, and the parametric bounded Löb axiom, producing long intermediate states that exhaust the output budget. We also note one mathematically interesting edge case that we accept as a valid outcome: `MirrorBot` vs `MirrorBot` does not terminate for any fuel offset, because both sides recursively simulate the other against itself. The agent correctly proved `outcome` n `MirrorBot MirrorBot = none`.

Discussion. The completeness on Tiers 0 and 1 (32/32) indicates that whenever the proof obligation is reducible by Lean's evaluator alone, the retrieve-propose-check pattern is highly reliable. Difficulty concentrates on `.search`-using bots, where the proof must invoke the proof-oracle axioms; in our setup, the empirical bottleneck is output budget rather than search depth or library coverage. The median of 2 tool-use iterations across the full matrix also indicates that the iterative compiler-feedback loop is doing real work: a single-shot configuration would discard the partial proofs the agent recovers after the first compile error.

4.2. Nash Equilibria of the Program Meta-game

We complement the Lean-side automation of §3.3 with a computational study of the equilibrium structure of the Lean 4 bot library. Concretely, we fix the eight admissible programs that admit a well-defined self-play outcome[3],

$$\Sigma_{\text{lib}} = \{\texttt{CooperateBot}, \texttt{DefectBot}, \texttt{DBot}, \texttt{EBot}, \texttt{OBot}, \texttt{TitForTatBot}, \texttt{CupodBot}\ \texttt{DupocBot}\}.$$

and enumerate every (mixed) Nash equilibrium of the meta-game $G^{\text{prog}}_{\text{lib}}$ obtained by restricting the program-equilibrium of Definition 2.1 to this strategy set, which are identified with the PE of the underlying PD.

Setup. The bimatrix is constructed from the action outcomes of each ordered bot pair formally derived in our (truncated) Lean library for 63 of the 64 cells[4] and the PD payoffs $(T, R, P, S) = (3, 2, 0, -1)$. Extreme NE are enumerated via the labelled best-response-polytope approach of Avis et al. (2010) executed in exact rational arithmetic through `lrsnash` from `lrslib` (Avis et al., 2010).

Result. The enumeration yields exactly 16 extreme Nash equilibria, partitioned into 5 *Nash components* in the sense of Jansen (1981) which are listed in Table 4, Appendix C.

[3]As stated in the failure analysis of §4.1, `MirrorBot` does not terminate when played against itself so we exclude it from the analysis.

[4]The (`CupodBot`, `DupocBot`) cell is fixed by configuration to (C, D), which corresponds to the conjectured outcome in Critch et al. (2022).

The striking, non-classical feature is the *range* of equilibrium outcomes that the open-source enlargement of the strategy space exhibits; while the underlying PD has a unique Nash equilibrium (D, D) with payoff 0 and joint cooperation rate 0, the program meta-game admits Nash equilibria realizing the cooperative payoff $R = 2$ and joint cooperation rate exactly 1. In particular, there exists a component which consists *entirely* of full-cooperation equilibria, with every profile in this component yielding payoff 2 and $\Pr[(C, C)] = 1$. Three further components support stable mutual cooperation at rates $11/18 \approx 0.61$, $14/27 \approx 0.52$, and $43/147 \approx 0.29$, with payoffs $3/2$, $4/3$, and $6/7$ respectively. The base-game equilibrium (D, D) persists as the $\Pr[(C, C)] = 0$ component.

The role of provability-conditioned cooperation. The `DupocBot` construction is essential for the appearance of these fully cooperative equilibria. Every extreme NE with $\Pr[(C, C)] = 1$ has `DupocBot` in the support of *both* the row and the column mixed strategy, and removing `DupocBot` from the library Σ_{lib} destroys the full-cooperation Component entirely. The provability-conditioned design of `DupocBot` is the unique program whose presence supports fully cooperative Nash equilibria; it turns the Pareto-efficient cooperative outcome, which is ruled out in equilibrium in the underlying PD, into an attained equilibrium payoff of the program meta-game.

For more details on these results, see Appendix C.

5. Conclusion

We presented the first Lean 4 formalization of proof-based Open Source Game Theory, populated with nine bots and the outcome theorems governing how they play against one another, mechanizing in particular the provability-conditioned constructions of Critch et al. (2022). On top of this verified library, we built an agentic pipeline that turns a natural language strategy description into a machine-checked outcome theorem, closing 40 of 45 pairs in the hand written bot matrix; the remaining failures concentrate exactly where the formal complexity does, on the `.search`-using bots, and we view closing this gap as the principal direction for future work. As a first application, we used the verified outcomes to enumerate the Nash equilibria of the resulting program meta-game and identified `DupocBot` as the unique element of the library whose presence is necessary for full mutual cooperation to arise as a rational phenomenon.

Acknowledgments

This work was supported in part by the German Federal Ministry of Education and Research (BMBF): Tübingen AI Center, FKZ: 01IS18039B; by the Machine Learning Cluster of Excellence, EXC number 2064/1 – Project number

390727645; by Coefficient Giving; by SwissAI; by Schmidt Sciences SAFE-AI Grant; by the Frontier Model Forum and AI Safety Fund;by the Canadian AI Safety Institute Research Program at CIFAR; by the Canadian AI Safety Institute Research Program at CIFAR through a Catalyst Award; by the Survival and Flourishing Fund; by the Cooperative AI Foundation; and by the Division of Engineering Science at the University of Toronto. Resources used in preparing this research project were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute.

References

Anthropic. Claude Opus 4.7, 2026. URL https://www.anthropic.com/news/claude-opus-4-7. April 16, 2026.

Avis, D., Rosenberg, G. D., Savani, R., and Von Stengel, B. Enumeration of nash equilibria for two-player games. *Economic theory*, 42(1):9–37, 2010.

Bárász, M., Christiano, P. F., Fallenstein, B., Herreshoff, M., LaVictoire, P., and Yudkowsky, E. Robust cooperation in the prisoner's dilemma: Program equilibrium via provability logic. *CoRR*, abs/1401.5577, 2014. URL http://arxiv.org/abs/1401.5577.

Bereska, L. and Gavves, S. Mechanistic interpretability for AI safety - A review. *Trans. Mach. Learn. Res.*, 2024, 2024. URL https://openreview.net/forum?id=ePUVetPKu6.

Chlipala, A. *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN 978-0-262-02665-9. URL http://mitpress.mit.edu/books/certified-programming-dependent-types.

Cooper, E., Oesterheld, C., and Conitzer, V. Characterising simulation-based program equilibria. In Walsh, T., Shah, J., and Kolter, Z. (eds.), *Thirty-Ninth AAAI Conference on Artificial Intelligence, Thirty-Seventh Conference on Innovative Applications of Artificial Intelligence, Fifteenth Symposium on Educational Advances in Artificial Intelligence, AAAI 2025, Philadelphia, PA, USA, February 25 - March 4, 2025*, pp. 13735–13744. AAAI Press, 2025. doi: 10.1609/AAAI.V39I13.33501. URL https://doi.org/10.1609/aaai.v39i13.33501.

Critch, A. A parametric, resource-bounded generalization of löb's theorem, and a robust cooperation criterion for open-source game theory. *Journal of Symbolic Logic*, 84 (4):1368–1381, 2019. doi: 10.1017/JSL.2017.42. URL https://doi.org/10.1017/jsl.2017.42.

Critch, A., Dennis, M., and Russell, S. Cooperative and uncooperative institution designs: Surprises and problems in open-source game theory. *CoRR*, abs/2208.07006, 2022. doi: 10.48550/ARXIV.2208.07006. URL https://doi.org/10.48550/arXiv.2208.07006.

Hardin, G. The tragedy of the commons. *Science*, 162 (3859):1243–1248, 1968. doi: 10.1126/science.162.3859.1243.

Harsanyi, J. C. Games with incomplete information played by "bayesian" players, i-iii part i. the basic model. *Management Science*, 14(3): 159–182, November 1967. doi: 10.1287/mnsc.14.3.159. URL https://ideas.repec.org/a/inm/ormnsc/v14y1967i3p159-182.html.

Harsanyi, J. C. Games with incomplete information played by "bayesian" players part ii. bayesian equilibrium points. *Management Science*, 14 (5):320–334, January 1968a. doi: 10.1287/mnsc.14.5.320. URL https://ideas.repec.org/a/inm/ormnsc/v14y1968i5p320-334.html.

Harsanyi, J. C. Games with incomplete information played by 'bayesian' players, part iii. the basic probability distribution of the game. *Management Science*, 14 (7):486–502, March 1968b. doi: 10.1287/mnsc.14.7.486. URL https://ideas.repec.org/a/inm/ormnsc/v14y1968i7p486-502.html.

Jansen, M. Maximal nash subsets for bimatrix games. *Naval research logistics quarterly*, 28(1):147–152, 1981.

Kovarík, V., Oesterheld, C., and Conitzer, V. Game theory with simulation of other players. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, pp. 2800–2807. ijcai.org, 2023. doi: 10.24963/IJCAI.2023/312. URL https://doi.org/10.24963/ijcai.2023/312.

Kovarík, V., Oesterheld, C., and Conitzer, V. Recursive joint simulation in games. *CoRR*, abs/2402.08128, 2024. doi: 10.48550/ARXIV.2402.08128. URL https://doi.org/10.48550/arXiv.2402.08128.

Lin, Y., Tang, S., Lyu, B., Yang, Z., Chung, J., Zhao, H., Jiang, L., Geng, Y., Ge, J., Sun, J., Wu, J., Gesi, J., Lu, X., Acuna, D., Yang, K., Lin, H., Choi, Y., Chen, D., Arora, S., and Jin, C. Goedel-prover-v2: Scaling formal theorem proving with scaffolded data synthesis and self-correction. *CoRR*, abs/2508.03613, 2025. doi: 10.48550/ARXIV.2508.03613. URL https://doi.org/10.48550/arXiv.2508.03613.

Mennen, A. Prisoner's dilemma tournament results, 2013. URL https://www.lesswrong.com/posts/QP7Ne4KXKytj4Krkx/prisoner-s-dilemma-tournament-results-0. LessWrong.

MixedMatched. Formalizing Game Theory in Lean. https://github.com/MixedMatched/formalizing-game-theory, 2023. GitHub repository. Accessed: 2026-05-24.

Nash, J. F. Equilibrium points in n-person games. *Proceedings of the national academy of sciences*, 36(1):48–49, 1950.

Nash, J. F. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

Nipkow, T., Paulson, L. C., and Wenzel, M. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 3-540-43376-7. doi: 10.1007/3-540-45949-9. URL https://doi.org/10.1007/3-540-45949-9.

Oesterheld, C. Robust program equilibrium. *Theory and Decision*, 86(1):143–159, 2019.

Osborne, M. J. and Rubinstein, A. *A course in game theory*. MIT press, 1994.

Ostrom, E. *Governing the Commons: The Evolution of Institutions for Collective Action*. Cambridge University Press, Cambridge, 1990.

Schelling, T. C. *The Strategy of Conflict*. Harvard University Press, Cambridge, MA, 1960.

Sistla, S. and Kleiman-Weiner, M. Evaluating llms in open-source games. *CoRR*, abs/2512.00371, 2025. doi: 10.48550/ARXIV.2512.00371. URL https://doi.org/10.48550/arXiv.2512.00371.

Song, P., Yang, K., and Anandkumar, A. Lean copilot: Large language models as copilots for theorem proving in lean. In Pappas, G. J., Ravikumar, P., and Seshia, S. A. (eds.), *International Conference on Neurosymbolic Systems, 28-30 May 2025, University of Pennsylvania, Philadelphia, Pennsylvania, USA*, Proceedings of Machine Learning Research, pp. 144–169. PMLR, 2025. URL https://proceedings.mlr.press/v288/song25a.html.

Tennenholtz, M. Program equilibrium. *Games Econ. Behav.*, 49(2):363–373, 2004. doi: 10.1016/J.GEB.2004.02.002. URL https://doi.org/10.1016/j.geb.2004.02.002.

Thakur, A., Tsoukalas, G., Wen, Y., Xin, J., and Chaudhuri, S. An in-context learning agent for formal theorem-proving. *arXiv preprint arXiv:2310.04353*, 2023.

Wang, H., Unsal, M., Lin, X., Baksys, M., Liu, J., Santos, M. D., Sung, F., Vinyes, M., Ying, Z., Zhu, Z., Lu, J., de Saxcé, H., Bailey, B., Song, C., Xiao, C., Zhang, D., Zhang, E., Pu, F., Zhu, H., Liu, J., Bayer, J., Michel, J., Yu, L., Dreyfus-Schmidt, L., Tunstall, L., Pagani, L., Machado, M., Bourigault, P., Wang, R., Polu, S., Barroyer, T., Li, W., Niu, Y., Fleureau, Y., Hu, Y., Yu, Z., Wang, Z., Yang, Z., Liu, Z., and Li, J. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *CoRR*, abs/2504.11354, 2025. doi: 10.48550/ARXIV.2504.11354. URL https://doi.org/10.48550/arXiv.2504.11354.

Wang, R., Zhang, J., Jia, Y., Pan, R., Diao, S., Pi, R., and Zhang, T. Theoremllama: Transforming general-purpose llms into lean4 experts. In Al-Onaizan, Y., Bansal, M., and Chen, Y. (eds.), *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024, Miami, FL, USA, November 12-16, 2024*, pp. 11953–11974. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.EMNLP-MAIN.667. URL https://doi.org/10.18653/v1/2024.emnlp-main.667.

Xin, H., Guo, D., Shao, Z., Ren, Z., Zhu, Q., Liu, B., Ruan, C., Li, W., and Liang, X. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *CoRR*, abs/2405.14333, 2024. doi: 10.48550/ARXIV.2405.14333. URL https://doi.org/10.48550/arXiv.2405.14333.

Yang, K., Swope, A. M., Gu, A., Chalamala, R., Song, P., Yu, S., Godil, S., Prenger, R. J., and Anandkumar, A. Leandojo: Theorem proving with retrieval-augmented language models. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023.

A. The Lean Engine: Detailed Definitions

This appendix provides the full definitions referenced in §3.1. All Lean source is reproduced verbatim from the `PrisonersDilemma` library; line-level changes are limited to whitespace.

A.1. Programs and Formulas

The `Prog` type is the syntactic class of strategies and the `Formula` type is the language the proof oracle quantifies over. They are mutually recursive: `.search` carries a `Formula` as its guard, and `.plays` takes `Prog`s as subjects: agents reason about agents reasoning about agents.

```
mutual
  inductive Prog where
    | const  : Action -> Prog
    | self   : Prog
    | opp    : Prog
    | bot    : Prog -> Prog
    | sim    : Prog -> Prog -> Prog
    | ite    : Prog -> Action -> Prog -> Prog -> Prog
    | search : Nat -> Formula -> Prog -> Prog -> Prog
  inductive Formula where
    | plays : Prog -> Prog -> Action -> Formula
    | impl  : Formula -> Formula -> Formula
    | neg   : Formula -> Formula
    | box   : Nat -> Formula -> Formula
end
```

Each constructor of `Prog` has the following informal reading:

Constructor	Meaning
`.const a`	Always play action a (C or D), ignoring the opponent.
`.self`	Placeholder for "my own source code."
`.opp`	Placeholder for "the opponent's source code."
`.bot p`	Closed reference to bot p — substitution does not descend inside.
`.sim p q`	Simulate program p against opponent q; return the action p plays.
`.ite g a p q`	Run g; if it returns action a, evaluate p, else evaluate q.
`.search k φ p q`	If φ is derivable in $\leq k$ steps, run p, else q.

On the `Formula` side, `.plays p q a` is the atomic predicate "program p plays action a against opponent q," `.box k φ` is the bounded provability operator $\Box_k\varphi$ ("φ is derivable in $\leq k$ proof steps"), and `.impl`, `.neg` are implication and negation (from which the remaining connectives are derivable).

Self-reference is resolved by mutually recursive substitution operators `Prog.subst` and `Formula.subst`, which traverse a program (resp. formula) and replace `.self` and `.opp` by concrete programs while leaving `.bot p` opaque:

```
mutual
  def Prog.subst : Prog -> (me opp : Prog) -> Prog
    | .const a,        _,  _   => .const a
    | .self,           me, _   => me
    | .opp,            _,  opp => opp
    | .bot p,          _,  _   => .bot p
    | .sim p q,        me, opp => .sim (p.subst me opp) (q.subst me opp)
    | .ite g a p q,    me, opp => .ite (g.subst me opp) a
                                       (p.subst me opp) (q.subst me opp)
    | .search k phi p q, me, opp =>
        .search k (phi.subst me opp) (p.subst me opp) (q.subst me opp)

  def Formula.subst : Formula -> (me opp : Prog) -> Formula
    | .plays p q a, me, opp => .plays (p.subst me opp) (q.subst me opp) a
    | .impl phi psi, me, opp => .impl (phi.subst me opp) (psi.subst me opp)
```

```
    | .neg phi,      me, opp => .neg (phi.subst me opp)
    | .box k phi,    me, opp => .box k (phi.subst me opp)
end
```

`Formula.subst` resolves `.self` / `.opp` occurrences inside the embedded `Prog` arguments of `.plays`, so that formulas appearing under `.search` are closed at evaluation time.

The semantic reading of a `Formula` is a Lean proposition:

```
def Formula.interp : Formula -> Prop
  | .plays p q a   => exists n, play n p q = some a
  | .impl phi psi  => phi.interp -> psi.interp
  | .neg phi       => not phi.interp
  | .box k phi     => proofSearch k phi = true
```

A.2. Evaluation and Outcomes

The proof-search oracle. Before defining the evaluator, we declare the oracle it consults. Conceptually, `proofSearch` $k\,\varphi$ should return `true` iff our ambient theory T has a proof of φ of size at most k. As discussed in §3.1, we take the axiomatic route rather than committing to a specific T: `proofSearch` is declared as an opaque `Bool`-valued function on $(\mathbb{N}, \texttt{Formula})$, and its intended behavior is pinned down by the axioms of §A.3.

```
noncomputable axiom proofSearch : Nat -> Formula -> Bool
```

Three points are worth noting. First, `proofSearch` is `noncomputable`: it stands for a decision procedure that we do not implement, so any definition consulting it inherits this status. Second, the formula passed to `proofSearch` must be closed, in particular, free of the placeholders `.self` and `.opp`, which is why the evaluator calls `Formula.subst` before each oracle invocation. Third, the `Nat` budget argument k acquires its meaning only through axiom A3 (§A.3), which ties `proofSearch k` to the existence of a witness of size at most k.

The evaluator. The evaluator is structurally recursive on a fuel parameter `n : Nat`, and is `noncomputable` because it consults `proofSearch`. The two helpers `play` and `outcome` are derived from `eval`.

```
noncomputable def eval : Nat -> Prog -> Prog -> Prog -> Option Action
  | 0,     _,   _,   _              => none
  | n+1,   _,   _,   .const a       => some a
  | n+1,   me,  opp, .sim p q       =>
      eval n (p.subst me opp) (q.subst me opp) (p.subst me opp)
  | n+1,   me,  opp, .ite g a p1 p2 =>
      match eval n me opp g with
      | some a' => if a' = a then eval n me opp p1 else eval n me opp p2
      | none    => none
  | n+1,   me,  opponent,   .bot p         => eval n me opponent p
  | n+1,   me,  opponent,   .self          => eval n me opponent me
  | n+1,   me,  opponent,   .opp           => eval n me opponent opponent
  | n+1,   me,  opp, .search k phi p1 p2 =>
      if proofSearch k (phi.subst me opp)
      then eval n me opp p1
      else eval n me opp p2

noncomputable def play (n : Nat) (p q : Prog) : Option Action := eval n p q p

noncomputable def outcome (n : Nat) (p q : Prog) : Option (Action x Action) := do
  let a <- play n p q
  let b <- play n q p
  some (a, b)
```

The library's outcome theorems are universal in n with a concrete fuel offset k:

$$\forall n,\ \texttt{outcome}\,(n+k)\,A\,B \;=\; \texttt{some}\,(a_1, a_2).$$

This shape is invariant to the fuel chosen at the call site once $n + k \geq$ the evaluation depth of A vs. B.

COMPUTABILITY OF EVALUATION

As declared above, `eval` is `noncomputable`, inherited from the opaque `proofSearch` oracle it consults at each `.search` guard. Because a reviewer may reasonably ask whether this reflects a fundamental obstruction, we make the situation precise: the noncomputability is *relative to our axiomatization*, not an absolute limit of bounded provability.

Why the oracle is not computable today. The block is best seen through two predicates. The first is "there exists a `ProofWitness` of size $\leq k$ proving φ"—the right-hand side of A3. Bounding k makes this a *finite* search, and it is in this bounded sense that proof search is decidable in principle; this is precisely the move by which Critch et al. (2022) collapses the unbounded, recursively-enumerable provability tower of Bárász et al. (2014) into a fixed-length check. The second predicate is what `proofSearch` must actually return `true` on. The cooperative Löb-fixpoint outcomes (canonically `CupodBot` against `DupocBot`) enter the oracle's specification through A8 (PBLT), which asserts `proofSearch` $m\,(\varphi\,k) =$ `true` for large k *without exhibiting a witness*, since `proofSearch` is itself `noncomputable`. A terminating enumeration of bounded proof terms therefore finds nothing for these outcomes, yet A3 and A8 jointly require `true`: no total computable function can satisfy the specification. The obstruction is thus not undecidability of bounded provability, which is decidable by enumeration, but that the reflection axioms inject witness-free members into what the oracle must accept.

How the block lifts. Replacing the reflection axioms with constructive theorems over an explicit calculus S, in particular a constructive proof of the bounded Löb principle that *exhibits* a size-$\leq k$ proof term, would make every member of the accepted predicate carry a finite witness. The oracle `proofSearch` then collapses to a decidable enumeration over proof terms, and `eval` becomes totally computable, Löb fixpoints included. This is finite, ordinary metamathematics rather than an impossibility result: the bounded regime is exactly where Gödelian incompleteness does not bite. We therefore regard a fully explicit, computable evaluator as foundational future work rather than a barrier.

A.3. Axioms

The proof system T is encoded by an abstract `ProofWitness` type, the `proofSearch` oracle of §A.2, and eight propositional axioms (A1–A8) relating witnesses, formulas, and the oracle. We make no assumption about T beyond these axioms.

```
axiom ProofWitness : Type
axiom witnessChars  : ProofWitness -> Nat
axiom witnessProves : ProofWitness -> Formula -> Prop
```

The first four axioms fix the behavior of `proofSearch` as a bounded-provability decision procedure over `ProofWitness`.

A1 — Soundness. Anything a witness proves is semantically true.

```
axiom witness_sound :
  forall w phi, witnessProves w phi -> phi.interp
```

Justification. Any reasonable proof system should be sound; we encode this as an axiom rather than proving it because we deliberately do not model T's internal proof rules. Without soundness, the `.search` constructor would be free to execute branches predicated on falsehoods.

A2 — Completeness for `.plays` formulas. Atomic plays-facts witnessed by concrete evaluation traces are derivable.

```
axiom witness_complete_plays :
  forall p q a, (exists n, play n p q = some a) ->
    exists w : ProofWitness, witnessProves w (.plays p q a)
```

Justification. This is the converse direction to A1's soundness, restricted to the atomic layer: whenever the evaluator actually produces an action, the proof system can certify that fact. We need it because `.search` and `.sim` guards reason about what programs play, and the code-reading axioms A5–A6 conclude `.plays` facts that downstream proofs must feed back into further provability queries; without A2 those concrete plays-facts would be true but unprovable in T, severing the

chain. It is safe to assume rather than derive because `.plays p q a` unfolds to the decidable arithmetic statement "$\exists n$, $\texttt{play}\, n\, p\, q = \texttt{some}\, a$," for which a concrete evaluation trace at the witnessing n is itself a finite proof. We restrict completeness to this atomic family deliberately: full completeness of T is neither true (by Gödel) nor needed, so we assume only the fragment the library consumes.

A3 — Exact budget semantics for `proofSearch`.

```
axiom proofSearch_spec :
  forall k phi, proofSearch k phi = true <->
    exists w, witnessProves w phi /\ witnessChars w <= k
```

Justification. A3 is the central axiom: it is what makes `proofSearch` a *bounded*-provability oracle rather than an arbitrary Boolean. The forward direction (A1 is its semantic shadow) lets a positive guard result be cashed into a concrete witness whose size is controlled by k; the backward direction lets the library *discharge* a guard by exhibiting such a witness. Both are needed: the `.search` evaluation rule branches on `proofSearch k`, so every outcome theorem for a `.search`-using bot must, at some point, either produce or consume a size-$\leq k$ witness, and A3 is the only bridge between the oracle's Boolean and the witness world of A1–A2, A4. It is the defining property of bounded provability in the sense of Critch et al. (2022) ("derivable in $\leq k$ steps") and is exactly what the internalize route would establish by implementing `proofSearch` as a bounded enumerator over T-derivations and proving it correct. Crucially, the right-hand side quantifies over witnesses of bounded size, so it is decidable in principle (§A.2); A3 is therefore an interface specification of a decidable predicate, not a smuggled oracle for undecidable provability.

A4 — Witness transport across formula families. A witness for a smaller parameter survives an increase, provided its size already fits the larger budget.

```
axiom witness_transport_family :
  forall (Phi : Nat -> Formula) n k, n <= k ->
  forall w, witnessProves w (Phi n) ->
  witnessChars w <= k ->
    exists w', witnessProves w' (Phi k) /\ witnessChars w' <= k
```

Justification. A4 is the strongest of the four witness axioms and the one that makes the library's *uniform-in-budget* theorem shape provable: the outcome theorems quantify over fuel with a single offset, so their proofs must transport one witness across an entire formula family $\Phi : \mathbb{N} \to \texttt{Formula}$ rather than re-deriving a fresh witness at each budget k. A4 grants exactly this: a witness at parameter n remains valid at any $k \geq n$ once its size fits the larger budget, and without it each existential outcome theorem would have to instantiate k at the call site and re-prove the witness pointwise, defeating the parametric statements that PBLT (A8) and the bot zoo rely on. We note, however, that the axiom as stated is *more general than is sound*: for an arbitrary family Φ there is no reason a witness for $\Phi(n)$ should yield one for $\Phi(k)$. It is only true for the families the library ever instantiates it with: bounded-provability statements whose sole k-dependence is in the budget annotation, where increasing k weakens the claim and the same proof term carries over. The honest formulation would restrict Φ to this budget-monotone class and prove transport by induction on the (unchanged) derivation; we state the unrestricted version as an axiom only to avoid committing to a representation of T-derivations, and every use in the library falls within the sound fragment.

The next three axioms encode proof-theoretic capabilities of T that we do not derive from a model of its internals: T can read the source code of `.search` and `.sim` bodies and chain implications by hypothetical syllogism.

A5 — Code-reading for `.search` bodies. If an agent's literal source code is `.search k` ψ `(.const a) (.const b)`, then T derives that bounded provability of ψ implies the agent plays a.

```
axiom proof_system_verifies_search_branch :
  forall (k : Nat) (psi : Formula) (a b : Action) (me opponent : Prog),
    me = .search k psi (.const a) (.const b) ->
    exists m, proofSearch m
      (.impl (.box k (psi.subst me opponent))
             (.plays me opponent a)) = true
```

Justification. The implication is true by inspection of the agent's code: a successful budget-k proof of ψ makes `eval` take

the `.const a` branch and return a. We make it an axiom rather than deriving it because we do not model T's internal proof rules; Critch et al. (2022) uses the same step silently when applying PBLT (Theorems 3.4 and 3.7).

A6 — Code-reading for `.sim` bodies. The direct analogue of A5 for the `.sim` constructor: if an agent's body is `.sim p q`, then T derives that the agent's action equals the action of p' against q', where p', q' are the substituted bodies.

```
axiom proof_system_verifies_sim :
  forall (me p q opponent : Prog) (a : Action),
    me = .sim p q ->
    exists m, proofSearch m
      (.impl (.plays (p.subst me opponent) (q.subst me opponent) a)
             (.plays me opponent a)) = true
```

Justification. Same status as A5: provable by inspection of the `.sim` eval rule, taken as an axiom because T's internals are not modeled.

A7 — Hypothetical syllogism in T. If T derives $\varphi \to \psi$ and $\psi \to \chi$, then T derives $\varphi \to \chi$.

```
axiom proofSearch_impl_trans :
  forall (phi psi chi : Formula),
    (exists m, proofSearch m (.impl phi psi) = true) ->
    (exists m, proofSearch m (.impl psi chi) = true) ->
    exists m, proofSearch m (.impl phi chi) = true
```

Justification. A basic structural rule for any reasonable proof system. Used to chain A5 with A6 when constructing PBLT premises for cross-pairings (e.g. a `.search` bot facing a `.sim` opponent).

A8 — Parametric Bounded Löb Theorem (PBLT). If T derives, for every $k > k_1$, that bounded provability of $\varphi(k)$ within $f(k)$ steps implies $\varphi(k)$, then T proves $\varphi(k)$ outright for sufficiently large k.

```
axiom PBLT :
  forall (phi : Nat -> Formula) (f : Nat -> Nat) (k1 : Nat),
    (forall a b, a <= b -> f a <= f b) ->
    (exists c kHat, c > 0 /\
       forall k, k > kHat -> f k > c * Nat.log2 k) ->
    (forall k, k > k1 ->
       exists m, proofSearch m (.impl (.box (f k) (phi k)) (phi k)) = true) ->
      exists k2, forall k, k > k2 ->
        exists m, proofSearch m (phi k) = true
```

Justification. PBLT is the constructive bounded counterpart of Löb's theorem, formalized as Lemma 3.6 in Critch et al. (2022). It is the technical engine behind both `CupodBot` and `DupocBot`: cooperative behavior conditioned on a provability check terminates not because of a fixed-point construction, but because PBLT promotes "T derives the conditional" to "T derives the consequent for large enough budget." The asymptotic side condition $f(k) \succ O(\log k)$ ensures the budget grows fast enough to encode the meta-level reasoning.

Summary of the trust base. In addition to the three interface declarations (`ProofWitness`, `witnessChars`, `witnessProves`) and the oracle declaration `proofSearch`, the library rests on eight propositional axioms: A1–A4 fix the behavior of the oracle as a bounded-provability decision procedure, A5–A7 encode proof-theoretic capabilities of T that we do not derive from a model of its internals, and A8 (PBLT) is the constructive Löb principle of Critch et al. (2022). No further axioms are introduced anywhere in the library.

A.4. Implemented Bots

This appendix collects every bot currently in the `PrisonersDilemma` library. Each entry pairs a Python pseudo-code sketch and the actual Lean term as it appears in the library. In the pseudo-code, `simulate(p, q)` denotes running bot `p` against bot `q`, and `proof_search` is the bounded oracle of §3.1. In Lean, `.opp` and `.self` are the placeholders for the opponent's source code and the bot's own source code respectively, resolved by substitution at evaluation time (Appendix A).

We group the bots in three families: action-only bots (`CooperateBot`, `DefectBot`), simulation-based bots

(`MirrorBot`, `TitForTatBot`, `DBot`, `OBot`, `EBot`), and provability-based bots (`CupodBot`, `DupocBot`).

ACTION-ONLY BOTS

`CooperateBot`. Ignores its opponent and always cooperates.

```
def CooperateBot(opp):
    return C
```

```
def CooperateBot : Prog := .const Action.C
```

`DefectBot`. Ignores its opponent and always defects.

```
def DefectBot(opp):
    return D
```

```
def DefectBot : Prog := .const Action.D
```

SIMULATION-BASED BOTS

These bots condition on the opponent's behavior against a fixed simple probe (typically `CooperateBot` or `DefectBot`). In Lean, simulation is the `.sim` constructor and the conditional is `.ite g a p q`, which evaluates `p` if the guard `g` returns action `a` and `q` otherwise.

`MirrorBot`. Mimics what opponent does against itself.

```
def MirrorBot(opp):
    return simulate(opp, MirrorBot)
```

```
def MirrorBot : Prog :=
  .sim .opp .self
```

`TitForTatBot`. Cooperates iff its opponent cooperates against `CooperateBot`. This is a one-shot analogue of the classical iterated tit-for-tat: it rewards opponents who appear cooperative against a maximally trusting probe.

```
def TitForTatBot(opp):
    if simulate(opp, CooperateBot) == C:
        return C
    else:
        return D
```

```
def TitForTatBot : Prog :=
  .ite
    (.sim .opp (.bot CooperateBot))  -- run opp against CooperateBot
    Action.C                          -- if it returns C ...
    (.const Action.C)                 --   ... cooperate
    (.const Action.D)                 -- otherwise defect
```

`DBot`. (**Mennen, 2013**) Exploits opponents naive enough to cooperate against `DefectBot`, and otherwise cooperates. It is the simplest example of a bot that punishes blind cooperation.

```
def DBot(opp):
    if simulate(opp, DefectBot) == C:
        return D
    else:
        return C
```

```
def DBot : Prog :=
```

```
  .ite
    (.sim .opp (.bot DefectBot))       -- run opp against DefectBot
    Action.C                            -- if opp would cooperate
    (.const Action.D)                   --   exploit: defect
    (.const Action.C)                   -- else cooperate
```

OBot. (Mennen, 2013) Cooperates only with opponents who pass both probes: cooperating against `CooperateBot` and against `DefectBot`. The second clause filters out exploiters in the style of `DBot`.

```
def OBot(opp):
    if simulate(opp, CooperateBot) == C:
        if simulate(opp, DefectBot) == C:
            return C
        else:
            return D
    else:
        return D
```

```
def OBot : Prog :=
  .ite
    (.sim .opp (.bot CooperateBot))    -- first probe
    Action.C
    (.ite                               -- if opp coop'd with CB,
      (.sim .opp (.bot DefectBot))     --   probe again with DB
      Action.C
      (.const Action.C)                 --   coop'd with both: cooperate
      (.const Action.D))                --   coop'd only with CB: defect
    (.const Action.D)                   -- failed first probe: defect
```

EBot. (Mennen, 2013) A layered strategy: punish naive cooperators (those who cooperate against `DefectBot`), then reward unconditional or mirror-like cooperators.

```
def EBot(opp):
    if simulate(opp, DefectBot) == C:
        return D                      # exploit naive cooperator
    elif simulate(opp, CooperateBot) == C:
        return C                      # reward cooperator
    elif simulate(opp, MirrorBot) == C:
        return C                      # reward mirror
    else:
        return D
```

```
def EBot : Prog :=
  .ite
    (.sim .opp (.bot DefectBot))       -- probe 1: against DefectBot
    Action.C
    (.const Action.D)                   --   cooperates? exploit
    (.ite
      (.sim .opp (.bot CooperateBot)) -- probe 2: against CooperateBot
      Action.C
      (.const Action.C)                 --   cooperates? cooperate
      (.ite
        (.sim .opp (.bot MirrorBot))  -- probe 3: against MirrorBot
        Action.C
        (.const Action.C)               --   cooperates? cooperate
        (.const Action.D)))             --   else defect
```

PROVABILITY-BASED BOTS

These bots condition on a bounded proof-search over the opponent's source code, using the oracle and soundness axioms of §2.1 and Appendix A. In Lean, the `.search k φ p q` constructor evaluates `p` if a proof of φ of size at most k exists, and `q` otherwise.

`CupodBot`. (Critch et al., 2022) Cooperates by default, defecting only when bounded proof search produces a witness, of size at most k, that the opponent defects against it.

```
def CupodBot(k, opp):
    if proof_search(k, opp, "opp(CupodBot(k)) == D"):
        return D
    else:
        return C
```

```
def CupodBot (k : Nat) : Prog :=
  .search k
    (.plays .opp .self Action.D)      -- hypothesis: opp plays D against me
    (.const Action.D)                  -- proof found: defect
    (.const Action.C)                  -- otherwise: cooperate
```

`DupocBot`. (Critch et al., 2022) The dual of `CupodBot`: defects by default, cooperating only on a bounded proof that the opponent cooperates against it.

A.5. Worked Example: CupodBot vs. TitForTatBot

When playing `CupodBot` against `TitForTatBot`, no such proof exists for any budget — `TitForTatBot` will not defect first — so PBLT yields the existential outcome theorem:

```
theorem CupodBot_vs_TitForTatBot :
  exists k0, forall k, k > k0 ->
    exists fuel, outcome fuel (CupodBot k) TitForTatBot
                 = some (.C, .C)
```

The proof instantiates PBLT with $\varphi(k) :=$ `.plays` (`CupodBot` k) `TitForTatBot` $.C$ and $f(k) := k$, discharges the "f increasing and super-logarithmic" side conditions immediately, and uses A2 to convert the semantic claim "`TitForTatBot` does not defect first" into a witness for the antecedent. The full proof is approximately 40 lines of Lean and is reproduced in the source repository.

B. The Agentic Pipeline: Full Results and Limitations

This appendix details the results for the agentic proof pipeline (§4.1), shows the prompts and an example trace behind those numbers, and exposes the pipeline's limitations.

B.1. Full Results for the Agentic Proof Automation

We ran the proof-search agent on all $\binom{9}{2} + 9 = 45$ unordered pairs from the hand-written library of §3.2. Each pair is attempted independently; if the first attempt returns a `ProofSearchError` (the agent exits without producing a compiling Lean source within its iteration budget), we re-run it once with the output budget raised from 16,384 to 32,000 tokens. Table 2 reports, for each tier pair, the pass count out of total pairs and the mean/median number of tool-use iterations and wall-clock seconds. When a pair was retried, the entry reflects the second attempt; means and medians are taken over all pairs in the tier, including the five that ultimately failed.

Two patterns are visible in the table. First, every pair in tiers 0×0, 0×1, 0×2, and 1×1 passes (32 of 32 pairs not involving a `.search`-using bot on *both* sides plus the four mixed Tier-2-against-constant pairs), with median iteration counts of 1–2 and median times under 90 seconds. These are the proof obligations that reduce by Lean's evaluator alone, modulo small rewriting steps; the agent essentially needs to identify the right unfolding lemmas. Second, the 1×2 row is where pass-rate drops (6/10) and cost spikes by an order of magnitude in both iterations and wall-clock time. These are the

Table 2. Per-tier aggregate results for the LLM proof-search agent on the full bot matrix (one row per unordered pair; when a pair was retried, the second attempt is reported). Tier 0: constant; Tier 1: reactive (no `.search`); Tier 2: `.search`-using. Iterations and time are reported as mean / median over all pairs in the tier (including failures).

Tier pair	Passed	Total	Mean it.	Med. it.	Mean (s)	Med. (s)
0×0	3	3	1.0	1	10.9	10.9
0×1	10	10	2.1	2	27.1	24.4
0×2	4	4	2.5	2.5	87.5	89.4
1×1	15	15	2.5	2	96.1	77.2
1×2	6	10	7.3	7	526.7	426.3
2×2	2	3	3.0	1	487.2	364.4
Total	**40**	**45**	**3.4**	**2**	**196.1**	**77.2**

proofs that must chain the oracle axioms (soundness, completeness, witness transport, PBLT, and the code-reading axioms of Appendix A.3), and as discussed in §4.1, the empirical bottleneck on the failures is the per-call output budget rather than the agent running out of ideas. The single 2×2 failure is the `CupodBot` vs. `DupocBot` cell, which is the open problem of Critch et al. (2022) and is therefore expected to be unprovable in the library as it stands.

B.2. Prompts and Example Traces

This subsection exposes the artifacts behind the aggregate numbers of §B.1: the prompts driving the proof-search agent, and a representative successful trace.

Agent Prompts

The proof-search agent is a single-turn-per-iteration loop around a Claude model (Anthropic, 2026) with two tools. The system prompt is fixed across runs; the user prompt is templated per bot pair and includes retrieved few-shot theorems.

System prompt.

```
"""You are an expert Lean 4 proof assistant for the open-source game theory project.

# Library definitions

These are the exact source files that define the types, evaluator, and axioms you must
    use.
Do not invent definitions, use only what is shown here and imported in the existing
    theorem files.
-- Program.lean
```lean
{Program.lean}
```

-- Dynamics.lean
```lean
{Dynamics.lean}
```{Axioms.lean}

# Your task

Write a complete, compilable Lean 4 theorem file that proves the requested outcome
    theorem.
Use the `run_lean_proof` tool to check your proof. Read errors carefully and fix them.
Use the `read_library_file` tool to inspect existing bot definitions or existing proofs
    for guidance.

# Rules
- The file must compile with zero errors and zero warnings in stderr.
- Import only modules that exist in the PrisonersDilemma library.
- The namespace must be `PDNew.Theorems`.
- **Do NOT redefine bots in your proof file.** Every bot already lives in its own
```

```
  module under `PrisonersDilemma.Bots.*`, import it (e.g. `import PrisonersDilemma.Bots.
    CupodBot`)
  and reference it by name. The proof file must contain only theorems, no `def` of any
    bot.
  Redefining a bot causes a namespace clash at `lake build` time.
- Do not use `sorry`, `admit`, or `native_decide`.
- Prefer `unfold`, `simp`, `rfl`, `exact`, `rw`, `cases`, `omega` tactics.
- When you are confident the proof compiles cleanly, output the final Lean source inside
  a ```lean ... ``` code fence and say "PROOF COMPLETE"."""
```

User prompt template (per pair).

```
"""Prove the following outcome theorem:

```lean
theorem llm_outcome_{LEFT}_vs_{RIGHT} (n : Nat) :
    outcome (n+{FUEL}) {LEFT} {RIGHT}
      = some (.{LEFT_ACTION}, .{RIGHT_ACTION}) := by
  sorry  -- replace with a real proof
```

Use fuel offset `+{FUEL}` exactly.

Important: name your theorem exactly
`llm_outcome_{LEFT}_vs_{RIGHT}` to avoid clashing with
existing library theorems.

Bot definitions (for reference only -- DO NOT redefine
these in your proof file):

--- Bots/{LEFT}.lean ---
```lean
{LEFT_BOT_SOURCE}
```

--- Bots/{RIGHT}.lean ---
```lean
{RIGHT_BOT_SOURCE}
```

Use exactly these import lines in your proof file to
reference the bots:

```lean
import PrisonersDilemma.Bots.{LEFT}
import PrisonersDilemma.Bots.{RIGHT}
```

Known outcome theorems involving these bots:
{KNOWN_THEOREMS_SUMMARY}

Here are relevant existing theorem files for reference:

--- {FEWSHOT_FILE_1} ---
```lean
{FEWSHOT_SOURCE_1}
```

[... additional retrieved few-shot files ...]

Use the `run_lean_proof` tool to check your proof.
Iterate until it compiles cleanly, then output the final
source and say PROOF COMPLETE."""
```

SUCCESSFUL RUN

Pair: `TitForTatBot` vs. `CooperateBot`. Outcome: (`.C`, `.C`). Iterations: 2. Wall-clock: ~ 22 s.

Agent's generated proof.

```
import PrisonersDilemma.Program
import PrisonersDilemma.Dynamics
import PrisonersDilemma.Bots.CooperateBot
import PrisonersDilemma.Bots.TitForTatBot

open PDNew
open PDNew.Bots

namespace PDNew.Theorems

theorem llm_outcome_CooperateBot_vs_TitForTatBot (n : Nat):
    outcome (n+3) CooperateBot TitForTatBot = some (.C, .C) := by
  have hA : play (n+3) CooperateBot TitForTatBot = some .C := rfl
  have hB : play (n+3) TitForTatBot CooperateBot = some .C := by
    show eval (n+3) TitForTatBot CooperateBot TitForTatBot = some .C
    unfold TitForTatBot CooperateBot
    simp [eval, Prog.subst]
    decide
  simp [outcome, hA, hB]

end PDNew.Theorems
```

B.3. Limitations and Future Work

The current pipeline is a minimum viable implementation aimed at validating the retrieve-propose-check pattern on a custom inductive type. We deliberately leave several extensions to future work: (a) the natural-language-to-Lean step has no reviewer agent — translation errors propagate silently to the proof stage, where they manifest as proof failures rather than bot-correctness failures; (b) there is no automatic rewriter loop feeding proof failures back into the bot writer; (c) the retrieval process is elementary and could be enhanced with a semantic-based approach.

C. Enumeration of Program Equilibria of the 8-bot Library

This appendix complements §4.2. We give, in turn: (C.1) the exact restricted meta-game on which the enumeration is carried out; (C.2) the algorithm for enumeration of extreme Nash Equilibria; (C.3) the full table of extreme Nash equilibria, with explicit Nash-component decomposition.

C.1. The Restricted Meta-game

Let $\Sigma_{\text{lib}} = (\pi_1, \ldots, \pi_8)$ be the ordered list of admissible programs

$$(\texttt{CooperateBot}, \texttt{CupodBot}, \texttt{DBot}, \texttt{DefectBot}, \texttt{DupocBot}, \texttt{OBot}, \texttt{TitForTatBot}, \texttt{EBot}).$$

For each ordered pair $(\pi_i, \pi_j) \in \Sigma_{\text{lib}}^2$, the action outcome $\texttt{outcome}(\pi_i, \pi_j) \in \{C, D\}^2$ is either formally derived in our Lean library or, in the single cell $(\texttt{CupodBot}, \texttt{DupocBot})$ left open in Critch et al. (2022), fixed by configuration to (C, D); none of the 16 extreme NE we report touches this cell. The row-player payoff matrix is

$$A_{ij} \;=\; u_{\text{row}}(\texttt{outcome}(\pi_i, \pi_j)), \qquad u_{\text{row}}(D, C) = b, \quad u_{\text{row}}(C, C) = b - c, \quad u_{\text{row}}(D, D) = 0, \quad u_{\text{row}}(C, D) = -c,$$

with $(b, c) = (3, 1)$ so that $(T, R, P, S) = (3, 2, 0, -1)$. The game is encoded as the *symmetric bimatrix* $(A, A^\top)$ with the column player's payoff when row plays i and column plays j being A_{ji}. See Table 3 for the complete outcome matrix, as it is formalized in our Lean library.

Table 3. Outcome matrix for the 9-bot program meta-game. Entries report the row-player and column-player actions. Bot names are abbreviated: Coop = `CooperateBot`, Cupod = `CupodBot`, DBot = `DBot`, Def = `DefectBot`, Dupoc = `DupocBot`, O = `OBot`, TFT = `TitForTatBot`, Mir = `MirrorBot`, and E = `EBot`. Actions are Cooperate (C) or Defect (D); `DupocBot` vs `CupodBot` is the open problem specified in Critch et al. (2022); and `MirrorBot` has no outcome (see §4.1 for more details).

	Coop	Cupod	DBot	Def	Dupoc	O	TFT	Mir	E
Coop	(C, C)	(C, C)	(C, D)	(C, D)	(C, C)	(C, C)	(C, C)	(C, C)	(C, D)
Cupod	(C, C)	(D, D)	(C, C)	(D, D)	Open Problem	(D, D)	(C, C)	(D, D)	(C, C)
DBot	(D, C)	(C, C)	(D, D)	(C, D)	(C, C)	(C, D)	(C, D)	(C, C)	(C, D)
Def	(D, C)	(D, D)	(D, C)	(D, D)	(D, D)	(D, D)	(D, D)	(D, D)	(D, D)
Dupoc	(C, C)	Open Problem	(C, C)	(D, D)	(C, C)	(D, D)	(C, C)	(C, C)	(C, C)
O	(C, C)	(D, D)	(D, C)	(D, D)	(D, D)	(D, D)	(D, C)	(D, D)	(D, C)
TFT	(C, C)	(C, C)	(D, C)	(D, D)	(C, C)	(C, D)	(C, C)	(C, C)	(D, C)
Mir	(C, C)	(D, D)	(C, C)	(D, D)	(C, C)	(D, D)	(C, C)	none	(C, C)
E	(D, C)	(C, C)	(D, C)	(D, D)	(C, C)	(C, D)	(C, D)	(C, C)	(C, C)

Table 4. All 16 extreme Nash equilibria of the restricted 8-bot program meta-game, grouped by Nash component and ordered by decreasing $\Pr[(C, C)]$. Bot names are abbreviated: `CB` = `CooperateBot`, `DB` = `DefectBot`, `DupocB` = `DupocBot`, `OB` = `OBot`, `EB` = `EBot`, `TFT` = `TitForTatBot`.

row support	col support	$u = v$	$\Pr[(C, C)]$	row strategy (non-zero entries)
CB, DupocB, TFT	CB, DupocB, TFT	2	1	$2/3, 1/9, 2/9$
CB, DupocB, TFT	DupocB	2	1	$2/3, 1/9, 2/9$
CB, DupocB, TFT	DupocB, TFT	2	1	$2/3, 1/9, 2/9$
DupocB	CB, DupocB, TFT	2	1	1
DupocB	DupocB	2	1	1
DupocB	DupocB, TFT	2	1	1
DupocB, TFT	CB, DupocB, TFT	2	1	$1/3, 2/3$
DupocB, TFT	DupocB	2	1	$1/3, 2/3$
DupocB, TFT	DupocB, TFT	2	1	$1/3, 2/3$
CB, OB, EB	CB, OB, EB	$3/2$	$11/18$	$1/2, 1/3, 1/6$
CB, DupocB, OB, EB	CB, DupocB, OB, EB	$4/3$	$14/27$	$1/3, 1/9, 1/3, 2/9$
CB, DB, DupocB, OB, EB	CB, DB, DupocB, OB, EB	$6/7$	$43/147$	$2/7, 2/7, 1/21, 2/7, 2/21$
DB	DB	0	0	1
DB	DB, OB	0	0	1
DB, OB	DB	0	0	$2/3, 1/3$
DB, OB	DB, OB	0	0	$2/3, 1/3$

C.2. Algorithm

We enumerate extreme Nash equilibria of the (shifted, strictly positive) bimatrix $(A, A^\top)$ via `lrsnash` from `lrslib` (Avis et al., 2010) implementing the labelled best-response-polytope enumeration of Avis et al. (2010) in exact rational arithmetic. The constant positivity shift of §C.1 leaves the Nash equilibrium set exactly invariant, so the output extreme NE are extreme NE of the original game.

C.3. Table of all Extreme Program Equilibria

Table 4 lists the 16 extreme Nash equilibria, grouped into the 5 Nash components of the restricted meta-game $G^{\text{prog}}_{\text{lib}}$. Mixed strategies are supported on the bots indicated by the support sets in the first two columns; bots absent from the support set carry zero mass. The payoffs satisfy $u = v$ on every equilibrium by symmetry of the underlying PD and of the outcome matrix construction; $\Pr[(C, C)]$ is the joint cooperation probability under the equilibrium mixed profile, computed from the action-pair outcome matrix.

How to read Table 4. Each grouping of rows in Table 4 corresponds to a single Nash component, which in our setting is a maximal Nash subset in the sense of Jansen (Jansen, 1981): a polytope of equilibria of product form

$$\mathcal{C} = X \times Y \;\subseteq\; \Delta(\Sigma_{\text{lib}}) \times \Delta(\Sigma_{\text{lib}}),$$

where X and Y are convex polytopes of row- and column-mixed strategies whose extreme points are explicitly enumerated in the table. The rows of Table 4 grouped under one component are precisely the vertices of $\mathcal{C}$: a list of $|\mathrm{ext}(X)| \cdot |\mathrm{ext}(Y)|$ entries pairing each extreme row-strategy with each extreme column-strategy. Two consequences are essential for reading the table:

(a) *Convex closure.* For every component $\mathcal{C}$ and every choice of convex weights $\lambda_i \geq 0$, $\mu_j \geq 0$ with $\sum_i \lambda_i = \sum_j \mu_j = 1$, the profile $\left(\sum_i \lambda_i \xi_i,\ \sum_j \mu_j \eta_j\right)$ obtained from the vertices (ξ_i) of X and (η_j) of Y is itself a Nash equilibrium of $G^{\mathrm{prog}}_{\mathrm{lib}}$.

(b) *Component invariants.* The payoff $u = v$ and the joint cooperation probability $\Pr[(C, C)]$ are constant on each component; the values recorded in Table 4 are thus properties of $\mathcal{C}$ as a whole.

Two structural features of Table 4 bear on the program-equilibrium theory of Bárász et al. (2014); Critch et al. (2022).

(i) A non-classical spectrum of equilibrium cooperation rates. The underlying Prisoner's Dilemma G has a unique Nash equilibrium (D, D) with joint cooperation probability 0; the NE-induced image of $\Pr[(C, C)]$ in G is the singleton $\{0\}$. Lifting to the program meta-game strictly enlarges this image: the five components realize

$$\Pr[(C, C)] \ \in\ \left\{ 1,\ \tfrac{11}{18},\ \tfrac{14}{27},\ \tfrac{43}{147},\ 0 \right\},$$

covering the full range from mutual defection to full cooperation. This is the qualitative phenomenon that motivates open-source game theory: $G^{\mathrm{prog}}_{\mathrm{lib}}$ admits Nash equilibria realizing the Pareto-optimal payoff $R = 2$ that is structurally unattainable in equilibrium of G, alongside a strict hierarchy of partially-cooperative components that are themselves stable under unilateral deviation in Σ_{lib}. The cooperative payoff is obtained as a positive-measure set of equilibria of the meta-game.

(ii) Necessity of `DupocBot` for full cooperation. The intersection of supports over the vertices of the cooperative component $\mathcal{C}_1$,

$$\bigcap_{(\xi,\eta)\,\in\,\mathrm{vert}(\mathcal{C}_1)} \mathrm{supp}(\xi) \cap \mathrm{supp}(\eta),$$

equals $\{\texttt{DupocBot}\}$: every vertex of $\mathcal{C}_1$ assigns strictly positive mass to `DupocBot` on both sides, and by convex closure (a) the same holds for every point of $\mathcal{C}_1$. Restricting the strategy space to $\Sigma_{\mathrm{lib}} \setminus \{\texttt{DupocBot}\}$ therefore eliminates $\mathcal{C}_1$ entirely, and inspection of Table 4 shows that no other component supports a cooperation rate exceeding $11/18$.

These results are the population-level counterpart of the self-cooperation theorem $\texttt{DupocBot}(k)$ vs. $\texttt{DupocBot}(k) = (C, C)$ of Critch et al. (2022): the provability-conditioned design is the unique program in our library whose presence in the population is necessary for full mutual cooperation to arise as a Nash phenomenon of the program meta-game.