Improving LLM Code Reasoning via Semantic Equivalence Self-Play with Formal Verification

Anonymous ACL submission

Abstract

We introduce a self-play framework for semantic equivalence in Haskell, utilizing formal verification to guide adversarial training between a generator and an evaluator. The framework leverages Liquid Haskell proofs for validating equivalence and execution-based counterexamples for inequivalence, organized via a difficulty-aware curriculum. To facilitate this, we release **OpInstruct-HSx**, a synthetic dataset of ≈28k validated Haskell programs. Empirical experiments show that our evaluator transfers effectively to downstream tasks, achieving up to 13.3pp accuracy gain on EquiBench and consistent gains on PySecDB. Ablation studies on the SEQ-SINQ regimes indicate that while inequivalence supervision provides data volume, equivalence proofs are uniquely responsible for the model's reasoning capabilities. The entire training pipeline and dataset are publicly released on GitHub and Hugging Face respectively.

1 Introduction

The rise of large language models (LLMs) has reshaped how software can be generated and maintained. Despite models such as Codex and Qwen2.5-coder have demonstrated strong capabilities in producing functional code from natural language prompts (Murphy et al., 2024; Hui et al., 2024), their outputs often fail to preserve the intended program behavior beyond basic test coverage (Laneve et al., 2025; Nguyen et al., 2025; Wei et al., 2025). This gap raises a fundamental challenge: How can we design training that explicitly teaches models to reason about semantic equivalence between programs? Addressing this problem is critical not only for reliable code generation but also for downstream applications such as program optimization, automated refactoring, and vulnerability detection.

Current approaches largely rely on test suites, which are insufficient for capturing deep semantic properties and edge cases. To bridge this gap, we ground our framework in Haskell for three key reasons. First, its pure functional nature and strong static typing eliminate hidden state and side effects (Thompson, 2011) (See Appendix A for illustration), making equivalence reasoning mathematically tractable (Launchbury, 1993; Sestoft, 1997). Second, the Liquid Haskell ecosystem enables the generation of machine-checkable proofs via refinement types, making it possible to certify semantic equivalence for a subset of Haskell programs (Liquid Haskell Tutorial, 2025). This offers a source of formal supervision that is unavailable in languages such as Python or Java, where formal verification is much harder. Finally, training on underrepresented functional paradigms pushes the model beyond standard object-oriented patterns (van Dam et al., 2024; Giagnorio et al., 2025), encouraging deeper abstraction capabilities.

We propose a self-play framework for semantic equivalence, in which two specialized agents interact: Alice, a generator that produces variants of reference programs; and Bob, an evaluator trained to decide whether two programs are equivalent. The self-play loop alternates between program generation, verification through proofs or counterexamples, difficulty scoring, and fine-tuning of both agents. By framing the problem as a game between generator and evaluator, the system encourages progressively harder examples and deeper reasoning about semantics.

This work investigates three core questions regarding the utility of functional programming for LLM alignment. First, we examine the dynamics of self-play, asking whether an adversarial loop in a functional language can induce a progressive curriculum that improves semantic reasoning. Second, we evaluate cross-domain and cross-language transferability, assessing whether semantic reasoning skills acquired in Haskell generalize to zero-shot synthesis and vulnerability detection in broader

coding benchmarks. Finally, we perform a controlled ablation to determine the relative contributions of supervision signals, distinguishing between the effects of formal equivalence proofs and execution-based counterexamples on evaluator robustness.

2 Related Work

Determining semantic equivalence is extremely challenging in general and current LLMs often fail to recognise semantic equivalence in code (Laneve et al., 2025; Nguyen et al., 2025; Wei et al., 2025). By Rice's Theorem, determining if two arbitrary programs are semantically equivalent is generally undecidable (Rice, 1953). Traditional approaches rely on unit testing; however, even extended test suites, such as HumanEval+ and MBPP+ remain insufficient to guarantee correctness (Gren and Antinyan, 2017; Chioteli et al., 2021). Similarly, symbolic execution offers path-sensitive analysis but suffers from combinatorial state-space explosion as program complexity increases (Badihi et al., 2020).

To address these incompleteness issues, recent research has pivoted towards formal verification. In the LLM domain, frameworks like DeepSeek-Prover (Xin et al., 2024a,b; Ren et al., 2025) and Kimina-Prover (Wang et al., 2025) have demonstrated that fine-tuning models on self-generated proofs (e.g., in Lean) significantly enhances their verifiable reasoning capabilities. However, generating fully machine-checkable equivalence proofs for imperative languages like Python or C++ is beyond the reach of current tools except for trivial cases (Miceli-Barone et al., 2025).

Our approach bridges this gap by using Haskell and Liquid Haskell. Liquid Haskell embeds refinement types into the language (Vazou et al., 2014), allowing logical properties to be verified automatically via Satisfiability Modulo Theories (SMT) solvers (Diatchki, 2015; Jhala et al., 2020; Liquid Haskell Tutorial, 2025). This framework enables the construction of machine-checkable lemmas, using reflection and Proof by Logical Evaluation (PLE), to formally certify pointwise equality between candidate functions. This provides a deterministic, high-fidelity feedback signal for the self-play loop.

Self-play has historically driven breakthroughs in game-playing agents like AlphaZero (Silver et al., 2017) and OpenAI Five (OpenAI et al., 2019). In the coding domain, recent frameworks such as Sol-Ver (Lin et al., 2025) and AutoIF (Dong et al., 2025) adapt this by using model-generated unit tests and execution feedback to filter synthetic data, achieving significant gains on benchmarks like MBPP and IFEval.

Our work is most directly built upon the adversarial framework proposed by Miceli-Barone et al. (2025), known as the Semantic Inequivalence Game (SINQ). In their setup, a generator ("Alice") creates program variants and diverging inputs (counterexamples), while an evaluator ("Bob") attempts to detect inequivalence without seeing the generator's justification. This adversarial loop provides a scalable curriculum that improves semantic reasoning. We extend this paradigm by including Semantic Equivalence tasks (SEQ). While Miceli-Barone et al. (2025) focused exclusively on inequivalence via execution feedback, we integrate formal verification using Liquid Haskell. This allows us to train on positive instances of equivalence supported by reasoning traces.

3 Methodology

We introduce the core methodology of the self-play framework for improving code reasoning in LLMs through two complementary tasks: the SEQ and the SINQ. In the SEQ task, the generator model is asked to produce a functionally identical variant of a given Haskell program, along with a formal proof certifying its equivalence. In contrast, the SINQ task requires generating a function that diverges from the original program on at least one input. Figure 1 shows the complete self-play framework, with difficulty-based supervised fine-tuning that guides the adaptive training loop.

3.1 Framework Overview

The self-play framework is formulated as a two-agent adversarial game between Alice (the generator) and Bob (the evaluator).

1. Each round starts with a reference Haskell program P, upon which Alice's task is to generate program Q, which is either a semantically equivalent variant to P (with a proof) or a semantically inequivalent program (with a diverging input that demonstrates their different behaviour).

2. The proof or the diverging input is verified afterwards, and if it fails, Alice loses.

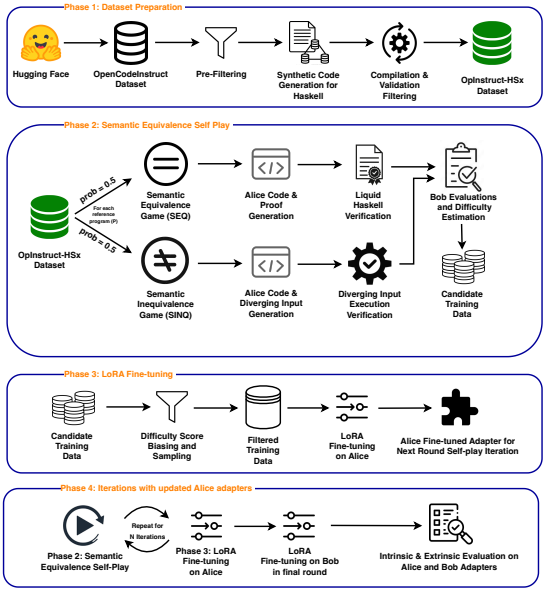

Figure 1: Overview of the semantic self-play framework for improving code reasoning in LLMs via Haskell.

3. Bob then decides whether (P, Q) are semantically equivalent. If Bob's judgment is accurate, he wins and Alice loses, vice versa.

Alice's objective is to create instances that are difficult for Bob to classify, whereas Bob's task is to correctly assess them. Through repeated interactions, both agents gradually improve their performance. Miceli-Barone et al. (2025) have proved that this adversarial framework has **no theoretical upper bound on model's performance improvement**, and in principle both agents can learn endlessly about the complex programming logic while training on a real-world coding dataset.

3.2 Dataset Generation and Preparation

The availability of high-quality Haskell datasets remains extremely limited. Among the few usable resources, the most substantial one is the Blastwind dataset[1], which aggregates real-world source files scraped from public GitHub repositories. However, its utility is hindered by substantial noise: the data contains unannotated, inconsistently formatted, and often non-compilable code.

To address the scarcity of high-quality Haskell datasets, we adopt a complementary strategy: **introducing OpInstruct-HSx, where we generate a synthetic Haskell dataset** by adapting from the `nvidia-OpenCodeInstruct` dataset (Ahmad et al., 2025), a large-scale, high-quality instruction corpus originally built for Python code generation.

The programs are first pre-filtered and then transformed into Haskell programs using the `DeepSeek-R1-Distill-Llama-70B` model. This process results in a synthetic dataset of Haskell programs. To ensure its quality, we applied an automated filtering and validation stage. For each generated Haskell program, we extract the function name and its argument types using syntactic heuristics, and synthesize a type-correct input using a recursive literal generator supporting common base types (e.g., `Int`, `Bool`, `List`, `Tuple`). Each program is compiled with Glasgow Haskell Compiler (GHC) and executed on the synthesised input. Only those that compile successfully and execute without errors are retained. This filtering process eliminates malformed or non-functional code, ensuring that the resulting dataset consists of minimally functional and executable Haskell programs.

Figure 2 shows the entire multi-stage filtering mechanism. We have contributed OpInstruct-HSx, a clean and executable Haskell dataset for both SEQ and SINQ games, which consists of approximately 28,000 validated Haskell functions derived from real-world problems. This dataset is made publicly available[2], serving as a high-quality synthetic Haskell resource for training LLMs in semantic reasoning tasks. The code to create the data and replicate the experiment is released on a public repository[3].

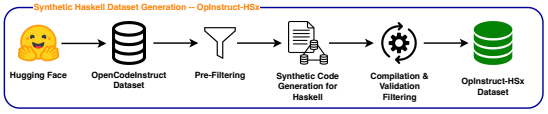

Figure 2: Full Pipeline for the **OpInstruct-HSx** dataset generation

3.3 The Self-Play Loop: Alice and Bob

Step 1: Program Selection and Branching

Let $\mathcal{D}$ be the dataset of reference Haskell programs. We randomly choose a reference program $P \in \mathcal{D}$ and then select the SEQ game with 50% probability, otherwise we choose the SINQ game.

[1] https://huggingface.co/datasets/blastwind/github-code-haskell-file

[2] URL redacted to preserve anonymity.

[3] URL redacted to preserve anonymity.

Step 2a: SEQ Game (Alice's Turn)

In the SEQ game, Alice receives P and must synthesize Q such that for all inputs x:

$$\forall x \in \mathcal{X}.\ P(x) = Q(x)$$

To challenge Bob, Alice is encouraged to construct a hard instance Q, aiming for a maximum target difficulty level (d=10, defined in Section 3.4.1). In addition, Alice is required to produce a formal proof (in Liquid Haskell) of this semantic equivalence. See Appendix B for an SEQ instance.

Step 2b: SINQ Game (Alice's Turn)

Alternatively in the SINQ game, Alice is instructed to produce a function Q that diverges from P on at least one input:

$$\exists x^* \in \mathcal{X} : P(x^*) \neq Q(x^*)$$

Alice is also incentivized to construct a difficult function Q that Bob is likely to misclassify, again targeting a maximum difficulty level (d=10). Alice must also output a diverging input x_a showing this inequivalence such that $P(x_a) \neq Q(x_a)$. See Appendix B for an SINQ instance.

Step 3: Verification through Liquid Haskell or Execution

- **SEQ Game**: Alice's proof is verified by Liquid Haskell, which acts as an external oracle. If the proof is accepted, the proof is retained as a fine-tuning example.
- **SINQ Game**: The candidate x_a is tested by an execution: If $P(x_a) \neq Q(x_a)$, the instance is accepted. Otherwise, the sample is discarded.

All candidates undergo compilation, execution, and formal verification checks. This ensures the training data for both Alice and Bob remains high-quality and executable.

Step 4: Bob's Turn – Difficulty Estimation

After Alice produces her candidate program Q, Bob is presented with both P and Q only and must decide whether the two programs are semantically equivalent. To estimate the challenge posed by each example, Bob is sampled N times and the proportion of correct responses from Bob is then used to compute a difficulty score. Further details on the difficulty-based curriculum and dataset sampling can be found in Section 3.4.

3.4 Implementation with Supervised Fine-Tuning with Difficulty Score

Given the practical challenges of reinforcement learning (Appendix C), we instead adopt rejection sampling supervised fine-tuning (SFT). We construct fine-tuning datasets by having Alice generate challenging programs and Bob learn from his own correct identifications. This semi-adversarial pipeline ensures Alice continually refines her ability to craft borderline-difficult SEQ or SINQ program pairs, while Bob steadily improves at its reasoning ability in semantic equivalence. Detailed prompt formats for both agents are provided in Appendix D.

3.4.1 Alice's Training Data Selection

After Alice generates the candidate pairs (P, Q), Alice's outputs are not immediately used for fine-tuning. Instead, for each pair, Bob is asked to evaluate their semantic relation multiple times (typically $N = 10$). The number of correct Bob responses N_{success} determines the **difficulty score** $\hat{d}$ as defined in Equation 1.

$$\hat{d} = d(P, Q) = 10 \times \left(1 - \frac{N_{\text{success}}}{N}\right) \quad (1)$$

However, most of Alice's early generations are trivial for Bob. Including all examples would flood the dataset with low-difficulty cases that Bob already solves easily. Following the design in Miceli-Barone et al. (Miceli-Barone et al., 2025), we only retain examples that are sufficiently challenging, as determined by the difficulty score $\hat{d}$. Hence, shown in Schema 2, we split all (P, Q) pairs into $\mathcal{D}_{Hard}$ and $\mathcal{D}_{Easy}$, where τ is a chosen difficulty threshold (e.g., $\tau = 5$).

$$\begin{aligned} \mathcal{D}_{Hard} &= \{(P, Q) \mid d(P, Q) > \tau\}, \\ \mathcal{D}_{Easy} &= \{(P, Q) \mid d(P, Q) \leq \tau\} \end{aligned} \quad (2)$$

Finally, Alice's training dataset is comprised of three SFT examples for each validated pairs (P, Q). The first example pairs the prompts and Alice's generation, and directly trains Alice to generate a challenging program Q, which helps improve Alice's ability to craft precise SEQ or SINQ code (Schema 3). SP and UP denote the system and user prompts conditioned on the reference program P and a target difficulty $d = 10$. O represents the model's raw output, which includes the chain-of-thought followed by the generated program Q (and

the diverging input x in the case of SINQ).

$$\begin{aligned}\text{SEQ} &= \left(\text{SP}_A^{eq},\ \text{UP}_A^{eq}(P, 10),\ O_A^{eq}\right)\\ \text{SINQ} &= \left(\text{SP}_A^{inq},\ \text{UP}_A^{inq}(P, 10),\ O_A^{inq}\right)\end{aligned} \tag{3}$$

We then select every hard example plus 20% as many easy ones sampled round-robin across integer difficulty bins to maintain a balanced curriculum.

The second example, designated as a "difficulty-prediction" instance, teaches Alice to self-assess the hardness of its own creations: by taking Alice's generated program Q along with the Difficulty Prediction User Prompt and supervising on the numeric label "Difficulty level:$\hat{d}$" (Schema 4). The training dataset is also biased towards hard examples, with easy examples comprising 50% of the subset. Alice learns to calibrate its difficulty estimates and ensures that its future generations are appropriately challenging.

$$\begin{aligned}\text{SEQ}_{\text{DP}} &= \left(\text{SP}_{A,DP}^{eq},\ \text{UP}_{A,DP}^{eq},\ \hat{d}\right)\\ \text{SINQ}_{\text{DP}} &= \left(\text{SP}_{A,DP}^{inq},\ \text{UP}_{A,DP}^{inq},\ \hat{d}\right)\end{aligned} \tag{4}$$

The third example concerns all (P, Q) pairs in the SEQ that have been successfully proved. We not only consider the generated candidate Q for a given P, but also the complete reasoning trace and valid Liquid Haskell proof script from Alice (Schema 5). These examples serve as supervised training signals to help Alice internalize the proof obligation π, corresponding to the refinement type $\forall x., P(x) = Q(x)$.

$$\text{SEQ}_{\text{proof}} = \left\{ \left(\text{SP}_A^{eq},\ \text{UP}_A^{eq}(P),\ O_A^{eq}(P, Q, \pi)\right) \;\middle|\; \text{LH}(P, Q) \vdash \pi \right\} \tag{5}$$

This three-part structure enables Alice to generate challenging variants, assess their difficulties, and internalize proof strategies.

3.4.2 Bob's Training Data Selection

Bob's training data is constructed from all of his correct (P, Q) pairs' evaluations. Each training example (Schema 6) consists of the original pair, the system prompts, user prompts, and Bob's response O_B. By sampling from a wide range of difficulties, Bob learns to reason and recognize both easy and subtle equivalence relations.

$$\mathcal{E}_{\text{Bob}} = \left(\text{SP}_B,\ \text{UP}_B(P, Q),\ O_B\right) \tag{6}$$

4 Experimental Setup

We use `DeepSeek-R1-Distill-Qwen-7B` as our base model for both Alice and Bob. Fine-tuning is performed using LoRA adaptors. Please see Appendix E for more details.

To investigate the distinct contributions of equivalence versus inequivalence supervision, we conduct the main experiment (E_0) alongside three controlled ablations (E_1–E_3). These regimes systematically vary the game type probability and reference program budget (P) to isolate the impact of SEQ supervision under controlled data constraints. The configurations are shown in Table 1.

Regime	SEQ/SINQ	P	Goal
E_0 (Base)	50/50	500	Main
E_1	0/100	500	Max Volume
E_2	96/4	500	Balanced Yield
E_3	0/100	40	Vol. Control ($E_3 \approx E_2$)

Table 1: Experimental regimes to test SEQ impact.

The full codebase is available at the GitHub repository[4] to generate the dataset OpInstruct-HSx, reproduce the experiments and evaluate model performance.

5 Results

The following evaluation results are structured to address the three research questions outlined in the introduction.

5.1 Intrinsic Evaluation

The goal of E_0 is teaching Alice (the generator) to generate increasingly challenging instances while Bob (the evaluator) to improve at judging program semantics.

5.1.1 Capability Assessment

As shown in Figure 3, the mean difficulty rises from 0.50 to 1.18 by round 7, indicating that Alice is crafting harder instances for a constant Bob. See Appendix H.1.1 for the difficulty trajectories breakdown for both SEQ and SINQ games.

In addition, we measure how much the evaluator model (Bob) improves after its first and only training round in the semantic equivalence self-play. Challenge instances are generated by the final trained generator model, Alice from round 7, using source programs from the training split of the OpInstruct-HSx and from the unseen test split. The

[4] URL redacted to preserve anonymity.

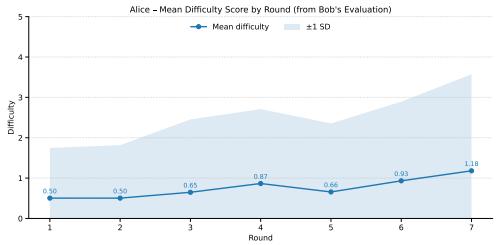

Figure 3: Mean and standard deviation for the difficulty scores for Alice's generated instances from a fixed untrained Bob over 7 rounds.

resulting accuracies, summarised in Table 2, show modest improvements on unseen data.

	Accuracy (%)		
Benchmark	**Base**	**Trained**	Δ
OpInstruct-HSx (Train)	87.57	**91.34**	+3.77
OpInstruct-HSx (Test)	88.24	**88.79**	+0.55

Table 2: OpInstruct-HSx accuracy results comparing the Base Model and Trained (Bob adapter) models.

5.2 Extrinsic Evaluation

Having validated the agents' improvement within the self-play loop, we now investigate whether this training enhances the model's understanding of program semantics across different programming languages and domains.

5.2.1 Haskell Program Generation

In MBPP and HumanEval from MultiPL-E (Cassano et al., 2022), both models' compilation errors decrease substantially and Pass@1 improves on the Haskell tasks (shown in Table 3). Performance trajectories for Alice across self-play rounds are provided in Appendix H.1.2 and H.1.3.

The results indicate that semantic reasoning training transfers positively to Haskell code generation performance and robustness for both agents. For Bob, the gains confirm that the semantic discrimination skills learned during self-play can yield broader benefits in practical programming scenarios. And for Alice, the improvements suggest that training on high-quality, verifiable program transformations in self-play can enhance downstream synthesis accuracy and reduce compilation errors.

5.3 Coding Related Tasks Evaluation

We would like to investigate whether richer type semantics learnt in the evaluator model Bob can further enhance his ability to analyse correct and semantically meaningful code across other coding paradigms, not just the functional ones.

5.3.1 EquiBench

The EquiBench (Wei et al., 2025) benchmark consists of six data categories: DCE (C pairs with dead/live code variations), x86-64 (identical assembly sequences via superoptimization), and CUDA (equivalent kernels with different scheduling) form the low-level systems group. The algorithmic group uses Python pairs from competitive programming, featuring OJ_A (algorithmic refactors), OJ_V (variable renaming), and OJ_VA (combined algorithmic and variable changes).

In the zero-shot prompting evaluation, our finetuned Bob model shows clear gains over the base model in several dataset configurations.

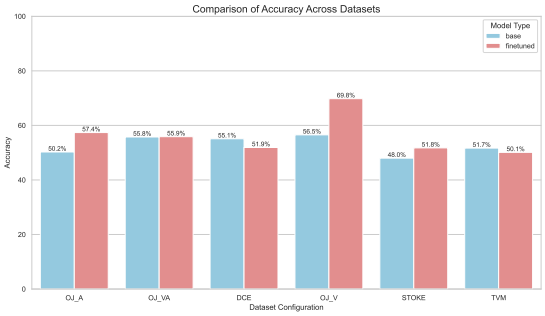

Figure 4: Accuracy comparison between base and finetuned Bob models across EquiBench dataset configurations.

Figure 5: F1 score comparison between base and finetuned Bob models across EquiBench dataset configurations.

From Figure 4 and Figure 5, the model transfers best to OJ_A and OJ_V because both categories (algorithmic refactors and variable renaming respectively) align with Bob's learned skill on detecting high-level behavioural divergence, resulting in large gains in performance. OJ_VA's performance stays modest as the authors stated that it is harder

Benchmark	Agent	Pass@1 (%)			Compilation Errors		
		Base	Trained	Δ	Base	Trained	Δ
HumanEval	Bob	17.7	**26.4**	+8.7	130	**110**	−20
HumanEval	Alice	17.7	**26.3**	+8.6	130	**104**	−26
MBPP	Bob	26.7	**36.9**	+10.2	250	**203**	−47
MBPP	Alice	26.7	**34.3**	+7.6	250	**218**	−32

Table 3: HumanEval and MBPP results (zero-shot prompting in Haskell), comparing the Base Model (untrained) and Trained models. Averages over 16 trials.

than pure renaming and closer to the "non-local structural" end.

In contrast, DCE, STOKE, and TVM require reasoning about low-level or non-functional semantics absent from our Haskell curriculum. This mismatch in language, abstraction level, and equivalence definition explains the weaker or negligible improvements on these tasks.

5.3.2 PySecDB

PySecDB is the first comprehensive dataset of security-related commits in Python (Sun et al., 2023). In Table 4, our fine-tuned Bob model achieves consistent improvements over the base model across all evaluated metrics. The results indicate that Bob's Haskell semantic reasoning training transfers positively to Python vulnerability detection, improving in identification of security-relevant code changes.

Metric	Scores (%)		
	Base Model	Trained	Δ
Accuracy	67.6	**68.8**	+1.2
Precision	48.0	**49.7**	+1.7
Recall	55.1	**59.2**	+4.1
F1 Score	51.3	**54.0**	+2.7

Table 4: PySecDB vulnerability detection results comparing the Base Model and Trained (Bob adapter).

5.3.3 CodeXGlue

CodeXGLUE's defect detection dataset (Zhou et al., 2019) is a curated collection of over 27,000 C-language functions, each manually labeled to indicate whether it contains a security-relevant defect.

In Table 5, both the base and fine-tuned Bob models perform near the 50% mark for accuracy, which is indistinguishable from a random model. CodeXGLUE's dataset gives Bob only one C function and asks if it's vulnerable. The vulnerabilities

Metric	Scores (%)		
	Base Model	Trained	Δ
Accuracy	48.8	**50.5**	+1.7
Precision	45.0	**45.7**	+0.7
Recall	**51.1**	41.7	−9.4
F1 Score	**47.9**	43.2	−4.7

Table 5: CodeXGLUE defect detection results comparing the Base Model and Trained (Bob adapter).

are low-level memory issues that require tracking pointers, memory allocation, and data layouts. However, Haskell is memory-safe, garbage-collected, and doesn't have manual frees or pointer arithmetic. Therefore, the trained model shows no meaningful advantage over the base model on this task.

5.4 Controlled Comparison of SEQ–SINQ Regimes

A critical finding from E_0 is the asymmetry in number of validated programs between SINQ and SEQ even under a uniform sampling policy in choosing the game. In Figure 6, we have hundreds of accepted SINQ instances per round, but very few SEQ instances (single digits in most rounds). A review of the output from the self-play shows that while Alice can readily produce SINQ pairs, the small reasoning model often struggles to produce Liquid Haskell proofs for SEQ, which drastically limits the number of compiled and verified SEQ examples available for training. This imbalance limits the diversity of program types in the training buffer, with the current SEQ+SINQ configuration heavily skewed with SINQ examples.

This subsection investigates whether incorporating SEQ supervision yields unique reasoning benefits beyond those of SINQ alone, aiming to isolate the impact of the supervision type from the disparity in verified data volume (see Appendix F for derivation details).

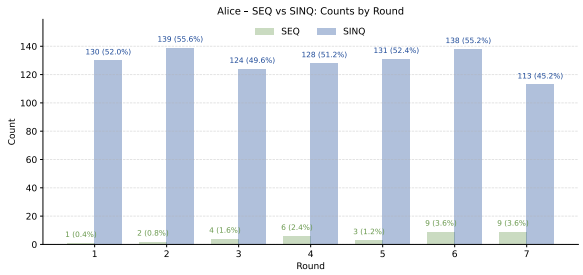

Figure 6: Counts and proportions of validated Alice generations by round.

5.4.1 Evaluation Results

We analyze the four experimental regimes to decouple the effects of supervision type (SEQ vs. SINQ) from training data volume. All results are presented in Appendix G and H.

Value of SEQ Supervision

Despite having significantly fewer verified training pairs in E_0 than in the pure SINQ regime E_1 (937 vs. $1,839$), E_0 achieves superior performance on semantic equivalence tasks. This indicates that SEQ supervision improves high-level semantic judgment even when data volume is reduced. Furthermore, when strictly controlling for data volume (E_2 vs. E_3, both ≈ 150 pairs), the inclusion of SEQ (E_2) yields consistent advantages on structural reasoning tasks over pure SINQ (E_3), shown in Table 7. This confirms that SEQ supervision confers a unique benefit that cannot be replicated by inequivalence signals alone.

Volume Trade-offs

While SEQ is qualitatively valuable, data volume remains critical. Regime E_2, which aggressively prioritizes SEQ (96% attempts) to force a balanced validated distribution, suffers a sharp drop in total verified pairs ($N = 140$) due to low proof yields, causing performance degradation across all benchmarks compared to E_0. The 50/50 mixed regime (E_0) therefore represents an acceptable trade-off, balancing the semantic depth of SEQ with the high verification yield of SINQ.

6 Conclusions

This paper investigates the use of self-play to improve code generation and semantic reasoning in LLMs, focusing on program equivalence in Haskell and Liquid Haskell. The framework introduces two agents: a generator of semantically equivalent or inequivalent program variants, and an evaluator trained to judge equivalence. Evaluation shows that Bob benefits substantially: Haskell coding performance improves, and transfer is strong on high-level semantic reasoning tasks, but negligible on low-level or memory-oriented tasks. Controlled comparisons reveal that although inequivalence data yields far more verified pairs, the inclusion of equivalence supervision confers unique gains in semantic reasoning that cannot be achieved through inequivalence alone.

7 Future Work

There are several promising avenues for future research. First, extending training across more rounds could allow Bob to accumulate richer experience. Additionally, (1) scaling to larger parameter models and (2) employing full-parameter fine-tuning instead of LoRA may increase both Alice's proof synthesis capabilities and Bob's reasoning capacity.

A second priority is to improve the statistical robustness of the controlled comparisons. The experiments in regimes E_2 and E_3 were limited by small sample sizes, which may not be representative of the true differences between SINQ- and SEQ-based supervision. Running larger-scale experiments would provide stronger evidence about the relative contributions of the two regimes.

Furthermore, methodologically speaking, the Haskell self-play framework could be expanded to cover low-level program behaviors by introducing tasks that explicitly model memory usage, side effects, and bit-level operations.

Moreover, converting the self-play pipeline into a dedicated Haskell Equivalence Evaluation test set would help to provide a more direct and sustainable benchmark for semantic reasoning, especially given the scarcity of high-quality Haskell resources currently available.

Finally, reinforcement learning could be re-examined as a means of enhancing the SEQ game's contribution. By running Alice long enough to generate sufficiently challenging programs and providing Bob with richer feedback through reward-based updates, it may be possible to overcome the current proof bottleneck and more effectively integrate equivalence reasoning into the self-play loop.

8 Limitations

A central limitation of this work lies in the proof synthesis bottleneck. Alice rarely produces Liquid

Haskell proofs that successfully pass PLE, resulting in very low validation yields for SEQ cases. This imbalance causes the majority of verified training data SINQ-dominated, reducing the contribution of SEQ supervision to Bob's learning. The issue is likely exacerbated by the relatively small model size used in this study, which likely restricts Alice's ability to generate structurally complex proofs.

Another limitation arises from the inherent constraints of Liquid Haskell itself. The verification process depends heavily on reflection and proof by logical evaluation, which are effective for local reasoning but struggle with certain classes of programs. Non-terminating behaviors, partial functions, and large-scale algebraic rewrites are difficult to certify, and in some cases impossible, within this fragment of the logic. Moreover, not all Haskell programs are reflectable, further narrowing the scope of tasks where equivalence can be formally verified.

Finally, there is a cross-domain mismatch in generalization. While Bob transfers strongly to high-level semantic reasoning tasks, such as OJ_A and OJ_V in EquiBench and vulnerability detection in PySecDB, his performance is notably weaker on low-level or stateful semantics, including DCE, STOKE, TVM, and CodeXGlue. This gap highlights a limitation of the current framework in addressing equivalence reasoning that depends on memory, side effects, or bit-level operations.

9 Ethical considerations

This work adversarially trains models to introduce hard to detect semantic modifications to computer programs and then to detect them, which may train them to detect security-relevant vulnerabilities or introduce obfuscated backdoors. In principle, these capabilities have a potential for malicious use, however, they can be also used for pro-social use to improve the reliability and security of code, whether generated by humans or LLMs. We believe that the net societal impact of models better able to reason about the subtleties of program semantics to be positive, especially as these capabilities are disseminated through Open Source releases, as we do in this work.

References

Wasi Uddin Ahmad, Aleksander Ficek, Mehrzad Samadi, Jocelyn Huang, Vahid Noroozi, Somshubra Majumdar, and Boris Ginsburg. 2025. Opencodeinstruct: A large-scale instruction tuning dataset for code llms. *Preprint*, arXiv:2504.04030.

Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. Ardiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 13–24.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *Preprint*, arXiv:2208.08227.

Efstathia Chioteli, Ioannis Batas, and Diomidis Spinellis. 2021. Does unit-tested code crash? a case study of eclipse. In *25th Pan-Hellenic Conference on Informatics*, PCI 2021, page 260–264. ACM.

Iavor S. Diatchki. 2015. Improving haskell types with smt. *SIGPLAN Not.*, 50(12):1–10.

Guanting Dong, Keming Lu, Chengpeng Li, Tingyu Xia, Bowen Yu, Chang Zhou, and Jingren Zhou. 2025. Self-play with execution feedback: Improving instruction-following capabilities of large language models. In *The Thirteenth International Conference on Learning Representations*.

Alessandro Giagnorio, Alberto Martin-Lopez, and Gabriele Bavota. 2025. Enhancing code generation for low-resource languages: No silver bullet. *arXiv preprint arXiv:2501.19085*.

Lucas Gren and Vard Antinyan. 2017. On the relation between unit testing and code quality. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, page 52–56. IEEE.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, and 1 others. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Ranjit Jhala, Eric Seidel, and Niki Vazou. 2020. Programming with refinement types: An introduction to liquid haskell. `https://ucsd-progsys.github.io/liquidhaskell-tutorial/book.pdf`. Version 13, July 20, 2020.

Cosimo Laneve, Alvise Spanò, Dalila Ressi, Sabina Rossi, and Michele Bugliesi. 2025. Assessing code understanding in llms. *Preprint*, arXiv:2504.00065.

John Launchbury. 1993. A natural semantics for lazy evaluation. In *POPL'9 3: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 144–154. ACM.

Zi Lin, Sheng Shen, Jingbo Shang, Jason Weston, and Yixin Nie. 2025. Learning to solve and verify: A self-play framework for code and test generation. *Preprint*, arXiv:2502.14948.

Liquid Haskell Tutorial. 2025. Liquid haskell tutorial : Introduction. https://liquid.kosmikus.org/01-intro.html. Accessed: 2025-07-18.

Yihao Liu, Shuocheng Li, Lang Cao, Yuhang Xie, Mengyu Zhou, Haoyu Dong, Xiaojun Ma, Shi Han, and Dongmei Zhang. 2025. Superrl: Reinforcement learning with supervision to boost language model reasoning. *Preprint*, arXiv:2506.01096.

Matéo Mahaut and Francesca Franzon. 2025. Repetitions are not all alike: distinct mechanisms sustain repetition in language models. *arXiv preprint arXiv:2504.01100*.

Antonio Valerio Miceli-Barone, Vaishak Belle, and Ali Payani. 2025. Program semantic inequivalence game with large language models. *arXiv preprint arXiv:2505.03818*.

William Murphy, Nikolaus Holzer, Feitong Qiao, Leyi Cui, Raven Rothkopf, Nathan Koenig, and Mark Santolucito. 2024. Combining llm code generation with formal specifications and reactive program synthesis. *arXiv preprint arXiv:2410.19736*.

Thu-Trang Nguyen, Thanh Trong Vu, Hieu Dinh Vo, and Son Nguyen. 2025. An empirical study on capability of large language models in understanding code semantics. *Information and Software Technology*, page 107780.

OpenAI, :, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison, David Farhi, Quirin Fischer, Shariq Hashme, Chris Hesse, Rafal Józefowicz, Scott Gray, Catherine Olsson, Jakub Pachocki, Michael Petrov, Henrique P. d. O. Pinto, Jonathan Raiman, and 8 others. 2019. Dota 2 with large scale deep reinforcement learning. *Preprint*, arXiv:1912.06680.

Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. 2022. Training language models to follow instructions with human feedback. *Preprint*, arXiv:2203.02155.

Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. 2025. Deepseek-prover-v2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. *Preprint*, arXiv:2504.21801.

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366.

Peter Sestoft. 1997. Deriving a lazy abstract machine. *Journal of Functional Programming*, 7(3):231–264.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *Preprint*, arXiv:1712.01815.

Shiyu Sun, Shu Wang, Xinda Wang, Yunlong Xing, Elisa Zhang, and Kun Sun. 2023. Exploring security commits in python. *Preprint*, arXiv:2307.11853.

Simon Thompson. 2011. *Haskell: the craft of functional programming*. Addison-Wesley.

Tim van Dam, Frank van der Heijden, Philippe de Bekker, Berend Nieuwschepen, Marc Otten, and Maliheh Izadi. 2024. Investigating the performance of language models for completing code in functional programming languages: a haskell case study. *Preprint*, arXiv:2403.15185.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement types for haskell. *SIGPLAN Not.*, 49(9):269–282.

Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, and 1 others. 2025. Kimina-prover preview: Towards large formal reasoning models with reinforcement learning. *arXiv preprint arXiv:2504.11354*.

Anjiang Wei, Jiannan Cao, Ran Li, Hongyu Chen, Yuhui Zhang, Ziheng Wang, Yuan Liu, Thiago SFX Teixeira, Diyi Yang, Ke Wang, and 1 others. 2025. Equibench: Benchmarking large language models' understanding of program semantics via equivalence checking. *arXiv preprint arXiv:2502.12466*.

Huajian Xin, Daya Guo, Zhihong Shao, Zhizhou Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. 2024a. Deepseek-prover: Advancing theorem proving in llms through large-scale synthetic data. *arXiv preprint arXiv:2405.14333*.

Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. 2024b. Deepseek-prover-v1.5: Harnessing proof assistant feedback for reinforcement learning and monte-carlo tree search. *Preprint*, arXiv:2408.08152.

Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. 2019. Devign: Effective vulnerability identification by learning comprehensive program

semantics via graph neural networks. In *Advances in Neural Information Processing Systems*, pages 10197–10207.

A Illustrative Haskell Features for Semantic Reasoning and Verification in LLM Self-Play

[1] Pure Semantics

Every function is a pure mapping from inputs to outputs, there is no hidden state or mutation. This purity ensures that two functions are semantically equivalent if they produce the same outputs for all inputs, regardless of how those outputs are computed.

```haskell
-- revRec :: [a] -> [a]
revRec :: [a] -> [a]
revRec []     = []
revRec (x:xs) = revRec xs ++ [x]
-- revFold :: [a] -> [a]
revFold :: [a] -> [a]
revFold = foldl (flip (:)) []

```

Because Haskell is referentially transparent, `revRec xs` can be substituted with `revFold xs` wherever it appears. This makes reasoning about equivalence both feasible and formalizable.

[2] Static Typing and GHC Compile

Haskell's static type system provides strong guarantees at compile time. Once a program is accepted by the compiler, most classes of errors such as type mismatches and null de-referencing are eliminated. This makes the type checker an effective pre-filter for program validity in the self-play loop.

```haskell
-- add :: Int -> Int -> Int
add x y = x + y

```

When writing `add True False`, GHC will raise a compilation error before running the program as `add` expects two `Int`.

B Liquid Haskell SEQ / SINQ Instances

Listing 1: Example of a SEQ Instance

```haskell
-- Reference Function P
double :: Int -> Int
double x = x + x

-- Alice's generated Q (Semantic Equivalent)
double_alt :: Int -> Int
double_alt x = 2 * x

-- Liquid Haskell Proof
{-@ lemma_double_equiv :: x:Int
      -> { double x == double_alt x } @-}
lemma_double_equiv :: Int -> Proof
lemma_double_equiv x
  =   double x
  === double_alt x
  *** QED
```

Listing 2: Example of a SINQ Instance

```haskell
-- Original function P
sign :: Int -> String
sign n
  | n < 0     = "negative"
  | n == 0    = "zero"
  | otherwise = "positive"

-- Alice's generated Q (semantically inequivalent)
signIneq :: Int -> String
signIneq n
  | n <= 0    = "non-positive"
  | otherwise = "positive"

-- Diverging input
x_a = 0
-- P x_a = sign 0 = "zero"
-- Q x_a = signIneq 0 = "non-positive"
```

C Implementation with Reinforcement Learning

Reinforcement learning (RL) offers a conceptually natural way to optimize the game by directly rewarding or penalizing generated programs based on the game outcome. A straightforward RL setup would proceed as follows based on the game rules:

C.1 RL Formulation

We treat Alice and Bob as two competing agents in a zero-sum Markov game.

- For the SINQ branch: An executor first checks that Q compiles and that $P(x_a) \neq Q(x_a)$ on Alice proposed diverging input x_a.
- For the SEQ branch: the SMT-based checker will execute the Liquid Haskell proof script asserting $\forall x.\ P(x) = Q(x)$.

If the above verification fails, the episode terminates with Alice receiving a negative reward $r_A = r_{\text{fail}} < 0$. Otherwise Bob observes state (P, Q) (but not Alice's diverging input x_a) and identify whether (P, Q) are semantically equivalent.

If Bob is correct, then Bob earns $r_B = r_{\text{success}} > 0$ and Alice is penalized with $r_A = r_{\text{too\_easy}} < 0$. If Bob is incorrect, he gets $r_B = r_{\text{fail}} < 0$ and Alice wins $r_A = r_{\text{win}} > 0$. Both agents update their policies to maximize expected cumulative reward over many self-play episodes.

C.2 Potential Benefits of RL

RL can be layered on top of the semantic equivalence self-play by treating Alice's generator as an RL agent whose policy is updated based on proof outcomes. Over many episodes, this encourages the policy to avoid classes of common semantic errors (e.g. off-by-one edge cases) and refines its internal value function to distinguish subtle equivalence-breaking patterns, resulting in an LLM that is both more precise and robust.

C.3 Practical Challenges

- **Sparse, High-Variance Rewards**: In our reasoning tasks, reward is only provided at the

end of a multi-step chain of thought. As a result, the LLMs often struggle to find successful trajectories, making it difficult for LLMs to learn which intermediate steps contributed to success (Liu et al., 2025).

- **Credit Assignment**: Attributing success or failure back to specific reasoning tokens or code-editing steps is nontrivial, and naive reward assignment can lead to undesirable shortcuts, such as inserting semantically neutral modifications to exploit reward signals without genuine reasoning.

- **Computational Resources Constraints**: RL is more computationally expensive than supervised fine-tuning due to the repeated sampling, model rollouts, and reward computation across multiple steps (Ouyang et al., 2022).

D Prompt Template

D.1 Alice SEQ System Prompt

```
You are a helpful and expert Haskell
    programmer, powered by Liquid Haskell.
Your job is to *transform* any given function
    `P` into a new function `Q` that:

  - Is syntactically correct Haskell.
  - Is semantically equivalent: ∀x. `P x == Q
    x`.
  - Uses a *different* implementation --
    Implement a non-structural change rather
    than merely swapping operator order.
  - Uses a different function name (e.g. add a
    trailing `'_alt`).

The original program and your program will be
    used in a test to evaluate the skill of an
    expert Haskell programmer who will
    identify whether your program Q is
    semantic equivalent to the original
    program P, so make sure that the
    difference you introduce are not very easy
    to understand.
You will be given a difficulty level from 0
    (easiest) to 10 (hardest) to target. E.g.
    difficulty level 0 means that an expert
    computer scientist in the bottom decile or
    above should be able to find a diverging
    example, difficulty level 9 means that
    only an expert computer scientist in the
    top decile should be able to find a
    diverging example, and difficulty level 10
    means that only the top 0.01 or less of
    expert Haskell programmer should be able
    to find a diverging example.

Always think through your transformation steps
    in `<think>...</think>`, then emit exactly:

Generated Program `Q`:
```haskell
<your Q here>
```
```

D.2 Alice SEQ User Prompt

```
Difficulty level: {difficulty_level}
Here is the original Haskell function `P`:
```haskell
{program_p_completion}
```

Its argument type is
```haskell
t = {t}
```

Your task: produce a new function `Q` that
    satisfies the system prompt requirements.
- Make sure `Q` has a different name (e.g.
    append a `'_alt`).
- Avoid trivial symmetric rewrites - show a
    genuine alternative implementation.
- Do not include any extra commentary beyond
    the required `<think>...</think>` and the
    `Generated Program `Q`: block.
- Where appropriate, feel free to use Prelude
    functions such as foldr, map, or zipWith
    to encourage diverse strategies.

<think>
```

D.3 Lemma SEQ Proof System Prompt

```
You are an expert Haskell/Liquid Haskell prover.
You are asked to prove that two reflected
    functions are equivalent.

The most basic proof should be in the following
    format:
```haskell
{{-@ lemma_{func_name_p}_equiv :: x:{arg_type}
    -> {{ {func_name_p} x == {func_name_q} x
    }} @-}}
        lemma_{func_name_p}_equiv :: {arg_type}
    -> Proof
        lemma_{func_name_p}_equiv x
            =.  {func_name_p} x
            === {func_name_q} x
            * QED
```

However, you should also use more advanced
    proof techniques if necessary.

Few-Shot Example 1:

```haskell
{{-@ LIQUID "--reflection" @-}}
{{-@ LIQUID "--ple" @-}}

module MyTest where

import Language.Haskell.Liquid.ProofCombinators

-- Alice program P
{{-@ reflect double @-}}
double :: Int -> Int
double x = x + x

-- Alice proposes Q
{{-@ reflect double' @-}}
double' :: Int -> Int
double' x = 2 * x

-- Here is the full lemma, from annotation to
    QED:
{{-@ lemma_double_equiv :: x:Int -> {{ double x
    == double' x }} @- }}
lemma_double_equiv :: Int -> Proof
lemma_double_equiv x
```
```

````
=   double x
=== double' x
* QED
```

Few-Shot Example 2:
```haskell
{{-@ LIQUID "--reflection" @-}}
{{-@ LIQUID "--ple" @-}}
module Equiv where

import Language.Haskell.Liquid.ProofCombinators

{{-@ reflect addNumbers @-}}
addNumbers :: Int -> Int -> Int
addNumbers a b = a + b

{{-@ reflect addNumbers' @-}}
addNumbers' :: Int -> (Int -> Int)
addNumbers' a = \b -> a + b

-- Alice detailed proof of equivalence
lemma_addNumbers_equiv :: Int -> Int -> Proof
lemma_addNumbers_equiv x y
    =   addNumbers x y
    === addNumbers' x y
    * QED

When you answer, output only the complete lemma
    block in the same style:
1. Use the `{{-@ lemma_... @-}}` annotation ,
    with the exact naming pattern
    lemma_<P>_equiv
2. The Haskell type signature
3. The function definition with `===` steps
4. End with `* QED`
5. Please put your proof between ```haskell and
    ```
No extra text, no additional comments.
Your answer must match the example format
    exactly, without trailing whitespace or
    newlines outside the code block.
````

D.4 Lemma SEQ Proof User Prompt

````
{error_msg_section}

{equiv_code}

--------------------------------------------------


Your task: Produce the proof of equivalence for
    the following function:
`{func_name_p} x == {func_name_q} x` for all
    `x`.

```haskell
{{-@ LIQUID "--reflection" @-}}
{{-@ LIQUID "--ple" @-}}
module Equiv where
import Language.Haskell.Liquid.ProofCombinators

{{-@ reflect {func_name_p} @-}}
{program_p_content}

{{-@ reflect {func_name_q} @-}}
{program_q_content}

-- Your complete proof of equivalence
/* PROOF BODY HERE */
```
<think>
````

D.5 Alice SINQ System Prompt

````
You are an expert Haskell programmer. Your task
    is to generate a semantically inequivalent
    variant of a given Haskell program, which
    means that there must exist at least a
    diverging input example such that the
    original program and your program either
    produce different outputs or exceptions,
    or one halts and the other one does not
    halt.
You must also provide a diverging input, which
    is a valid input for both programs, but on
    which they produce different outputs.

A good inequivalent program `Q` should be
    subtly different from `P`.
A good diverging input `x` should be simple and
    clearly demonstrate the semantic
    difference between `P` and `Q`.

The original program and your program will be
    used in a test to evaluate the skill of an
    expert Haskell programmer who will have to
    produce a diverging example (not
    necessarily the same as yours), so make
    sure that the difference you introduce are
    not very easy to understand.
You will be given a difficulty level from 0
    (easiest) to 10 (hardest) to target. E.g.
    difficulty level 0 means that an expert
    computer scientist in the bottom decile or
    above should be able to find a diverging
    example, difficulty level 9 means that
    only an expert computer scientist in the
    top decile should be able to find a
    diverging example, and difficulty level 10
    means that only the top 0.01 or less of
    expert Haskell programmer should be able
    to find a diverging example.

First, think step-by-step and write down your
    analysis of program `P` and your strategy
    for creating an inequivalent program `Q`.
    Enclose this reasoning within `<think>`
    and `</think>` tags.
After the thinking block, the final answer
    could only be in the following format,
    without any additional explanation or
    context.

Final output MUST be exactly:
Generated Program `Q`:
```haskell
<Your generated Haskell code for `Q`>
```


Diverging Input `x`:
```
<The diverging input `x`>
```
````

D.6 Alice SINQ User Prompt

````
Difficulty level: {difficulty_level}
Original program `P`:
```haskell
{program}
```


<think>
````

D.7 Bob System Prompt

````
You are an expert Haskell programmer. You are
    given two Haskell programs, `P` and `Q`.
Your task is to determine if they are
    semantically equivalent.
````
````

````
Use the following format to respond:
# Equivalent?
Yes or No

If the programs are equivalent, respond with
    your thought process and a final output
    with:
# Equivalent?
Yes

If they are inequivalent, respond with your
    thought process and a final output with:
# Equivalent?
No
````

D.8 Bob User Prompt

````
Program `P`:
```haskell
{program_p}
```

Program `Q`:
```haskell
{program_q}
```
<think>
````

D.9 Alice SEQ Difficulty Prediction System Prompt

````
Difficulty level: Any
Program P
```haskell
{program_p}
```
The program Q below is semantically equivalent
    to the original program P, where the
    equivalence is due to the fact that the
    two programs have the same behavior on all
    inputs.
Program Q
```haskell
{program_q}
```
````

D.10 Alice SINQ Difficulty Prediction System Prompt

````
Difficulty level: Any
program P
```haskell
{program_p}
```
The program Q below is semantically
    inequivalent to the original program P,
    where the inequivalence is due to the fact
    that the two programs have different
    behavior on some inputs.
Program Q
```haskell
{program_q}
```
````

D.11 Alice Difficulty Prediction User Prompt

```
Predict the difficulty level of the instance of
    Program Q compared to Program P. Just
    write \"Difficulty level: D\" where D is
    your prediction, do not write anything
    else.
```

E Main Experiment Setup

E.1 Model Configs and Experiment Settings

We employ `DeepSeek-R1-Distill-Qwen-7B` as the base model. The decoding parameters are configured as: temperature $T = 0.6$, top-$p = 0.95$, top-$k = 20$, presence penalty $= 1.5$ (Applied to reduce repetition in outputs from smaller reasoning models (Mahaut and Franzon, 2025)), and a context length of 32,768 tokens.

In E_0, Alice is configured with a 50/50 probability of playing either the SEQ or SINQ game (Section 3.3), with the difficulty level set as 10 in her prompts (Appendix D.2 and D.6). Bob serves solely as an evaluator and difficulty scorer, and he is not fine-tuned iteratively but only once after all rounds are complete.

E.2 Dataset Size and Difficulty Threshold

Due to computational constraints, each experimental run uses a maximum of 500 Haskell reference programs[5], i.e., $|\mathcal{D}| = P = 500$. The difficulty threshold τ is set to 3 (rather than the default 5) to increase the number of training examples available to Alice. For future experiments with greater resources, it is recommended to increase $|\mathcal{D}|$ and restore $\tau = 5$ to more closely simulate a fully adversarial setting.

E.3 Fine-Tuning Protocol

After each self-play round, Alice is fine-tuned on the combination of newly generated instances and the retained examples from all previous rounds. More importantly, each round's fine-tuning is initialized from the original base model rather than from the fine-tuned adapter checkpoint obtained in the previous round. This strategy is adopted to avoid bias accumulation across iterations and to maintain stable and unbiased difficulty estimation.

During generation in round $i + 1$, Alice uses the LoRA learned in round i, but fine-tuning for that round still begins from the base model.

E.4 Experimental Duration

The main run comprises 7 self-play rounds. In each round, Alice is fine-tuned according to the above protocol. Bob undergoes a single fine-tuning step after all 7 rounds are complete. For both Alice and Bob, the fine-tuning minimises the loss only on the model's own outputs.

[5] A main run with fine-tuning takes about 3 days on 4 NVIDIA L40S GPUs.

F Rationale and Derivations for the SEQ-SINQ Comparison Experiment

F.1 Fairness Criterion and Derivation

Let P be the number of reference programs attempted per round, $p \in [0,1]$ the probability of playing SEQ (so $1-p$ for SINQ), and let $r_{\text{SEQ}}, r_{\text{SINQ}}$ denote the verification yields (probability that an attempt becomes a verified training example). By linearity of expectation, the expected number of verified examples per round is

$$K(P,p) \;=\; P\big(p\, r_{\text{SEQ}} + (1-p)\, r_{\text{SINQ}}\big).$$

To isolate the effect of SEQ vs. SINQ unbalanced volume size, we match the expected count K of verified pairs across arms.

F.2 Yield Estimation from the Main Run E_0

From the 7-round of main experiment E_0, $|\mathcal{D}| = P = 500$ (per-round attempts ≈ 250 SEQ and 250 SINQ):

Game	Total Validated	Attempts	Yield ($\hat{r}$)
SEQ	34	1,750	1.94%
SINQ	903	1,750	51.6%

Table 6: Validation yields from E_0. Validated counts by round were: SEQ $(1,2,4,6,3,9,9)$ and SINQ $(130,139,124,128,131,138,113)$.

F.3 Predicting Supervision for Each Arm

To achieve a balanced distribution of verified training examples (approx. 50/50 split), we configured regime E_2 with a mix of 96% SEQ and 4% SINQ. With a reference budget of $P = 500$, the estimated verified yields per round are:

$$N_{\text{SEQ}} \approx 500 \cdot 0.96 \cdot 0.0194 \approx 9.3 \tag{7}$$

$$N_{\text{SINQ}} \approx 500 \cdot 0.04 \cdot 0.516 \;\; \approx 10.3 \tag{8}$$

Totaling these yields using $K(P,p)$ results in an expected $K \approx 19.6$ verified examples per round, achieving the desired parity between the two supervision signals.

F.4 Matched comparator for E_2

Choose P_{E_3} in a 100% SINQ run so that

$$\begin{aligned} P_{E_3} \cdot r_{\text{SINQ}} &= K_{E_2} \\ \Rightarrow P_{E_3} &= \frac{19.632}{0.516} = 38.04 \approx 40 \end{aligned} \tag{9}$$

F.5 What this controls, and what it does not

Matching K ensures each arm receives similar amounts of verified training, so outcome differences are attributable to the type of game (presence/absence of SEQ) rather than volume. However, residual differences may still arise from the difficulty distribution of accepted items and class-imbalance within Bob's updates.

F.6 Comparison Objectives

The experimental design supports the following four key pairwise comparisons:

- E_0 vs. E_1: Under fixed compute resources (constant P), does a mixed 50% SEQ / 50% SINQ regime yield greater performance improvements than a pure SINQ regime similar to Miceli et al. setting (Miceli-Barone et al., 2025), given that the latter produces substantially more verified training examples due to higher verification success rates in SINQ?
- E_0 vs. E_2: With fixed compute resources, does shifting towards a more balanced number of verified SEQ and SINQ examples, at the cost of reducing the overall number of verified training pairs, lead to improved evaluator performance compared to a SINQ-dominated main experiment?
- E_1 vs. E_2: Under the same compute constraints, does a pure SINQ regime (maximising verified training pairs) outperform a more balanced SEQ–SINQ dataset that sacrifices data volume for greater semantic diversity?
- E_2 vs. E_3 When the number of verified training pairs is held constant, does the inclusion of SEQ supervision yield measurable performance benefits over an SINQ-only regime?

F.7 Summary of experiments

- E_0 (main): 50/50, $P = 500$.
- E_1: 100% SINQ, $P = 500$.
- E_2: 96% SEQ / 4% SINQ, $P = 500$.
- E_3 (matched to E_2): 100% SINQ, $P = 40$ to equalize expected verified pairs with E_2.

This suite isolates the causal effect of adding SEQ supervision under controlled total verified pairs, enabling a fair assessment of whether SEQ contributes beyond SINQ alone.

G Experiment Results

Benchmark	Metric	Score (%)			
		$\mathbf{E_0}$	$\mathbf{E_1}$	$\mathbf{E_2}$	$\mathbf{E_3}$
HumanEval (Bob)	Pass@1	**26.4**	25.8	22.7	22.0
HumanEval (Alice)	Pass@1	26.3	**28.1**	21.8	18.4
MBPP (Bob)	Pass@1	36.9	**38.8**	34.7	32.4
MBPP (Alice)	Pass@1	34.3	**39.3**	32.9	28.0
EquiBench (OJ_A)	Acc	**57.4**	54.0	55.5	53.3
EquiBench (OJ_A)	F1	**50.7**	44.4	48.9	45.1
EquiBench (OJ_V)	Acc	**69.8**	61.4	63.2	65.8
EquiBench (OJ_V)	F1	**69.7**	58.0	61.5	64.7
EquiBench (OJ_VA)	Acc	**55.9**	54.5	54.7	53.7
EquiBench (OJ_VA)	F1	**47.5**	45.4	47.1	45.2
EquiBench (STOKE)	Acc	**51.8**	50.9	51.0	50.8
EquiBench (STOKE)	F1	**18.3**	15.4	16.8	15.9
EquiBench (TVM)	Acc	50.1	50.3	48.8	**50.8**
EquiBench (TVM)	F1	38.3	**39.9**	33.5	33.9
EquiBench (DCE)	Acc	51.9	**52.6**	51.5	51.3
EquiBench (DCE)	F1	**47.7**	45.4	45.2	44.4
PySecDB	Acc	**68.8**	67.3	68.7	68.7
PySecDB	F1	**54.0**	51.9	53.9	53.6
CodeXGlue	Acc	**50.5**	49.2	49.9	48.9
CodeXGlue	F1	43.2	44.0	44.1	**48.5**

Table 7: Benchmark Performances over all experiments. Averages over 16 samples.

Game	Count			
	$\mathbf{E_0}$	$\mathbf{E_1}$	$\mathbf{E_2}$	$\mathbf{E_3}$
SEQ	34	–	62	–
SINQ	903	1,839	78	156
Total	937	1,839	140	156

Table 8: Total verified generation from Alice of all experiments over 7 rounds.

H Extended Regime Results

H.1 Experiment E_0

Most of the evaluation results are included in the main report.

H.1.1 Difficulty trajectories for both task types

When we inspect the difficulty trajectories for both task types in Figure 7, they move upward across rounds.

- For SEQ, although the sample size is not statistically significant, its mean difficulty score jumps from near 0.0 in early rounds to $2.0 - 3.0$ in later ones, signaling that Alice manages to construct are non-trivial from Bob's perspective.
- For SINQ, its averages being greater than the medians (0.0 for all rounds) indicates a right-skewed distribution. This suggests that while most generated instances are of lower difficulty, Alice occasionally produces much harder examples that significantly influence the mean. Starting in round 4, the magnitude of the difference between the mean and median becomes larger, and by round 7 the mean even exceeds the upper quartile. Therefore, Alice is producing increasingly harder examples over successive rounds.

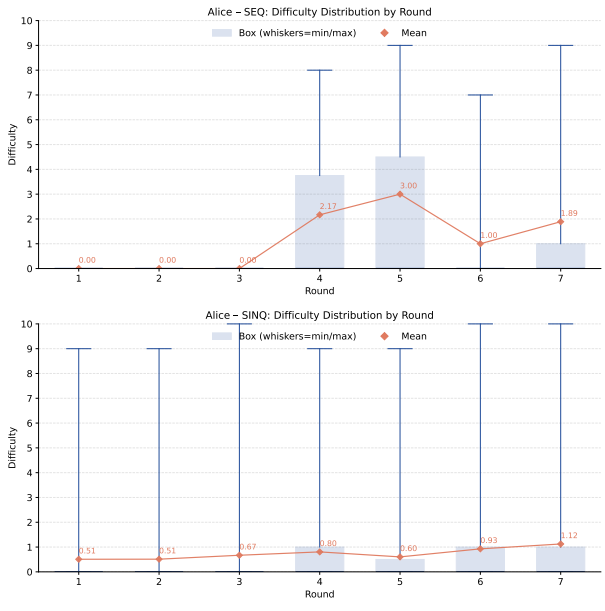

Figure 7: Mean and box-whiskey plots of the difficulty scores of Alice's generated instances by round. Top: SEQ Game. Bottom SINQ Game.

Both SEQ and SINQ setups show rising difficulty scores across rounds, indicating that Alice improves in generating more challenging instances for Bob.

H.1.2 Alice's HumanEval (Haskell) Performance

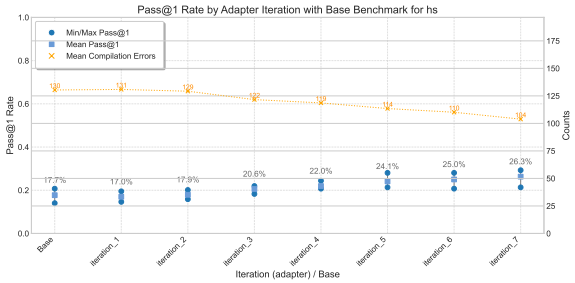

Figure 8: Alice's HumanEval results. Averages over 16 trials.

H.1.3 Alice's MBPP (Haskell) Performance

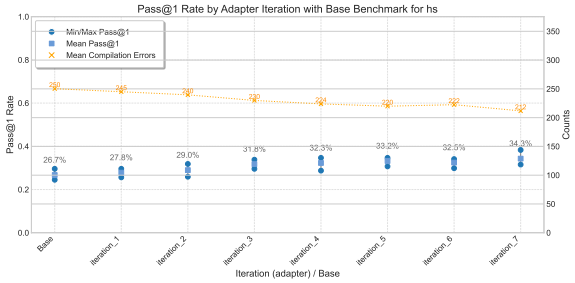

Figure 9: Alice's MBPP results. Averages over 16 trials.

H.2 Experiment E_1

H.2.1 Alice's HumanEval (Haskell) Performance

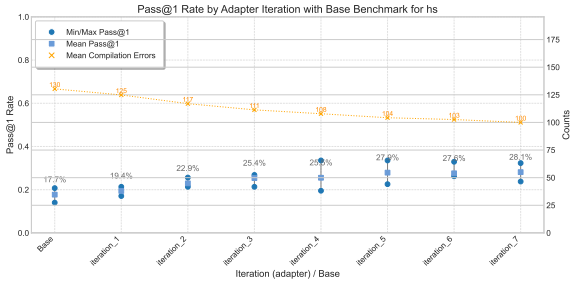

Figure 10: Alice's HumanEval results. Averages over 16 trials.

H.2.2 Alice's MBPP (Haskell) Performance

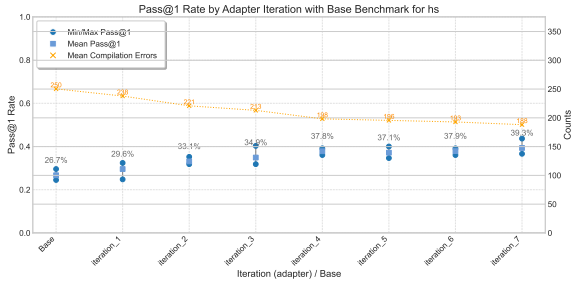

Figure 11: Alice's MBPP results. Averages over 16 trials.

H.2.3 Alice's SEQ vs SINQ Validated Generation Counts

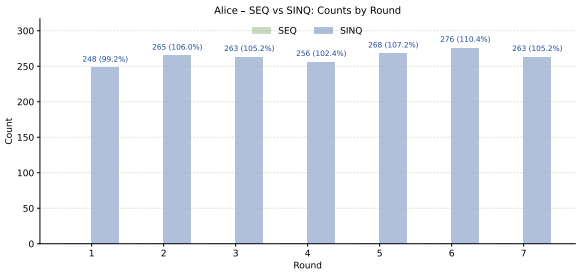

Figure 12: SEQ vs SINQ Validated Generation Counts.

H.3 Experiment E_2

H.3.1 Alice's HumanEval (Haskell) Performance

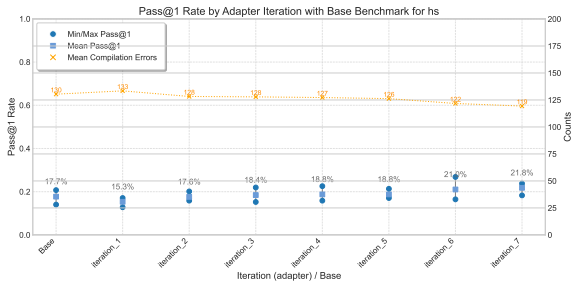

Figure 13: Alice's HumanEval results. Averages over 16 trials.

H.3.2 Alice's MBPP (Haskell) Performance

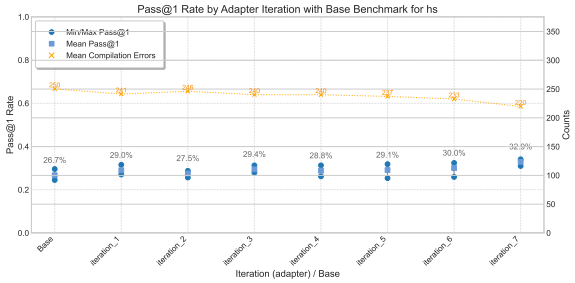

Figure 14: Alice's MBPP results. Averages over 16 trials.

H.3.3 Alice's SEQ vs SINQ Validated Generation Counts

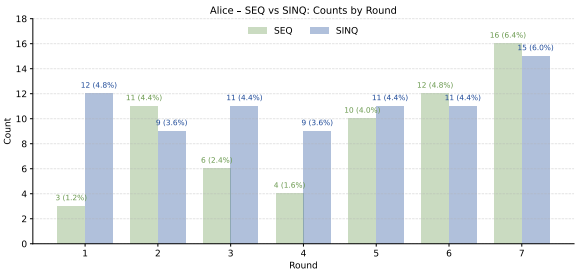

Figure 15: SEQ vs SINQ Validated Generation Counts.

H.4 Experiment E_3

H.4.1 Alice's HumanEval (Haskell) Performance

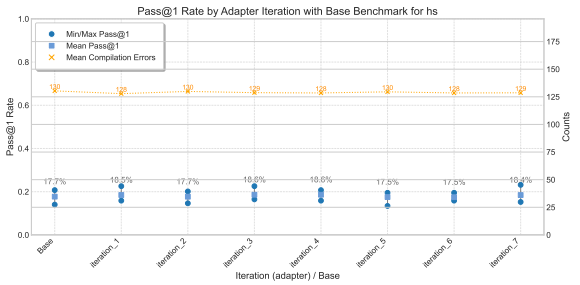

Figure 16: Alice's HumanEval results. Averages over 16 trials.

H.4.2 Alice's MBPP (Haskell) Performance

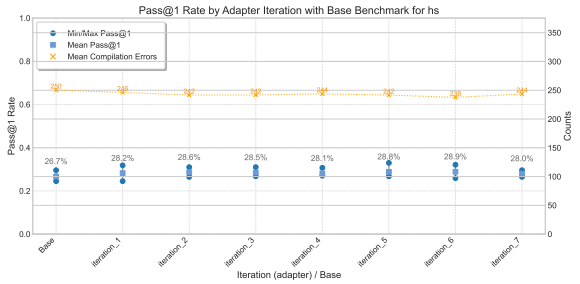

Figure 17: Alice's MBPP results. Averages over 16 trials.

H.4.3 Alice's SEQ vs SINQ Validated Generation Counts

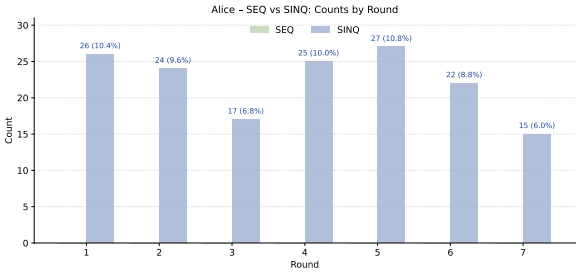

Figure 18: SEQ vs SINQ Validated Generation Counts.

H.4.4 E_2 vs E_3 Validated Generation Counts

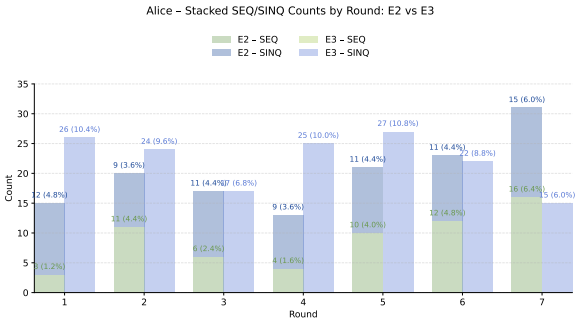

Figure 19: Validated Generation Counts of both experiment, indicating the effort of controlling P shown in Appendix F in order to achieve an equal number of verified generation.