FORMAL DISCO: SCALABLE OPEN-ENDED GENERATION OF FORMALLY VERIFIED PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

The cost of producing code is rapidly decreasing with increasingly capable AI agents, but quality assurance has not kept pace. Formal verification can provide strong correctness guarantees on generated code, but languages supporting verification are still challenging for coding models due to the scarcity of available examples of programs in those languages. To tackle this issue, we propose FORMAL DISCO: a scalable system for coordinated LLM-based workers that can be easily applied to open-ended synthetic data generation at scale. We use FORMAL DISCO to share tasks and programs between three classes of workers: "initiators", which read random READMEs from open-source repositories and documentation snippets to sketch a short related verified program, "fixers" which resolve issues given compiler/verifier feedback, and "extenders" that propose patches to expand working programs. FORMAL DISCO records and uses agent traces both for initial distillation from a stronger model and for self-improvement. To prevent diversity collapse, we propose a simple principle of maximum entropy for synthetic program generation, and use entropy maximization via iterative supervised fine-tuning to learn to generate increasingly diverse programs over time. We release large datasets of synthetic verified programs in three languages — Dafny, Verus, and Frama-C —, and fine-tune Qwen-2.5 Coder 32B for verification-relevant tasks, matching Claude Opus 4.5 on a Verus annotation task and closing much of the gap in all other cases. Overall, our work offers a path to using synthetic verified programs at scale to overcome the long-standing data barrier in AI-assisted verified programming.

1 INTRODUCTION

Formal program verification provides the strongest possible form of correctness assurance (Mugnier et al., 2025; Leino, 2010; Lattuada et al., 2023; 2024). A formally verified program comes with both a logical specification of its behavior as well as a proof that a given implementation satisfies the specification. If AI agents write verified programs, humans can check correctness by writing or inspecting a formal specification, but can then trust a symbolic verifier to judge whether the implementation satisfies the given specification.

Unfortunately, current LLMs, especially open-weight models, are still limited in their ability to write verified programs, or automate verification-related tasks (such as annotating loops with invariants). This is to be expected given the lack of human-written examples of programs in most languages supporting formal verification. For instance, while there are tens of millions of Python repositories on GitHub, the largest existing public dataset of human-written programs in Dafny is DafnyBench (Loughridge et al., 2024), with 782 Dafny files collected and deduplicated from public repositories.

In this paper, we tackle the data scarcity barrier that has hindered progress in AI for verified programming using LLMs themselves to generate synthetic data at scale, in an *open-ended fashion*. By open-ended, we imply that our primary goal is to generate a large volume of new, *diverse* verified programs in formal languages, including specifications, implementations, and proofs, without being directed a priori towards a particular downstream task or application. Ultimately, we then use these synthetic programs in a task-directed manner to train open models for verification-relevant tasks and evaluate them on tasks derived from human-written programs.

To this end, we introduce FORMAL DISCO, a language-agnostic framework for coordinating LLM-based workers around tasks and an object database, collecting traces for distillation and

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

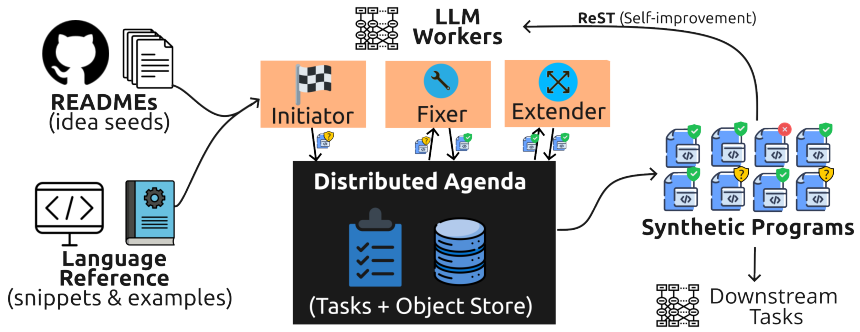

Figure 1: Overview of FORMAL DISCO. The system is centered around a distributed agenda, where LLM-based workers collaborate on completing tasks and store objects — here, synthetic formally verified programs being written, extended and repaired. The *Initiator* uses seed context to write initial programs. Programs might fail compilation or verification, and *Fixer* workers writes patches to repair them. Working programs grow in complexity by patches written by the *Extender* worker. Resulting synthetic data can be used to self-improve the workers and for training for downstream tasks.

self-improvement. To generate synthetic programs at scale, we implement three classes of agents in FORMAL DISCO: "initiators", that sample READMEs from random GitHub repositories and snippets from the language's documentation and sketch an initial program in a given language, "fixers" that take programs that fail compilation or verification and attempt to patch them, and "extenders" that generate patches to expand already working programs, making them grow in complexity.

Since we use the agent's own generated data to self-improve, diversity collapse can become an issue even though agents might improve at passing verification (Herel & Mikolov, 2024; Shumailov et al., 2024). We tackle this issue directly by proposing a conceptual framework of *entropy maximization* over program features for synthetic program datasets. This framework allows us to not only improve agents at generating programs that pass verification, but that are also *increasingly diverse* in entropy metrics derived from logical specifications, annotations, loop structures and proofs for lemmas.

We release a large dataset of programs in three verification-aware languages (Dafny, Verus and Frama-C), generated synthetically with FORMAL DISCO. Our dataset contains over 100k complete verified programs — for Dafny and Frama-C, this is orders of magnitude more programs than existing public datasets. Our framework based on feature entropy allows us to compare *datasets* of programs with each other, shedding light on their quality beyond just their raw size, which is a poor metric to assess datasets in general (Gupta et al., 2021). Finally, we train the open-weights Qwen-2.5 Coder 32B using our data for verification tasks in Dafny and Verus, and our synthetic datasets improves model performance on annotating human-written programs with logical assertions and invariants and on proving lemmas, often matching Claude 4.5 Opus on these tasks. Overall, FORMAL DISCO offers a language-agnostic path to overcome data scarcity in AI-assisted verified programming.

2 RELATED WORK

Formally verified code generation with LLMs. A growing line of work aims to address the reliability concerns of LLM program synthesis through formal verification Loughridge et al. (2024); Yang et al. (2025): having models produce not just code but machine-checkable specifications and proofs, whether in interactive theorem provers (e.g., Lean, Isabelle) or in auto-active verification languages such as Dafny Leino (2010) and Verus Lattuada et al. (2023), which target executable programs directly Misu et al. (2024); Sun et al. (2024).

Synthetic verified program generation. Recent work has also proposed synthetic data synthesis for tackling the data scarcity issue in AI for formal verification. **Verus.** Chen et al. (2026) describe the problem as multi-level data scarcity: there is a shortage not only of proofs, but also of *specifications* and even of Rust programs written in the Verus-supported subset of the language. Their dataset, SAFE, was constructed by translating tens of thousands of small Python and Rust programs from popular code-synthesis datasets into Verus-compatible Rust, and then generating specifications and proofs through inference and iterative fine-tuning of open models. The follow-up work on VeruSyn

(Di et al., 2026) used SAFE's same seed dataset and scaled up synthesis up to a dataset of 6.9M programs. VeruSyn expands the seed data distribution using "tutorial-based synthesis" to explicitly broaden coverage of Verus-specific features beyond what the seed data contains (SAFE, for instance, contained no instances of lemmas — known as "proof functions" in Rust — despite this being a commonly necessary feature for proving complex specifications). **Dafny**. There has also been parallel approaches for synthetic training data generation in Dafny, including ATLAS (Baksys et al., 2026), which focused on using seed programs in a higher-resource language, such as Python, and attempting to reimplement them into Dafny (which includes adding suitable specifications and proofs, not present in the original program), as well as the DafnySynth component in dafny-annotator (Poesia et al., 2024b), a fully synthetic dataset generated by scheduling GPT-4o to sketch, implement and extend new Dafny programs. Compared to these, FORMAL DISCO is the first synthetic program generation method that is shown to be language-agnostic — we apply it to Dafny, Verus and Frama-C, and adding new languages is simple. Moreover, FORMAL DISCO can directly optimize and improve the diversity of generated programs across iterations, overcoming the common worry that self-training LLMs on their own outputs can collapse diversity (Herel & Mikolov, 2024; Shumailov et al., 2024).

Synthetic data in formal mathematics. Beyond verified programming, synthetic data has driven recent progress in formal theorem proving, where proof checkers provide the same scalable correctness signal that program verifiers give us. AlphaGeometry (Trinh et al., 2024) reached near gold-medalist performance on olympiad geometry after pre-training on 100M synthetically generated theorems and proofs, while DeepSeek-Prover (Xin et al., 2024) and Goedel-Prover (Lin et al., 2025) grow large corpora of Lean proofs through cycles of autoformalization and proof search — a loop analogous to our distillation and self-improvement pipeline. These efforts, however, generate proofs for externally given (or translated) statements; FORMAL DISCO instead generates specifications, implementations and proofs jointly in an open-ended setting, as has been explored in formal mathematics (Poesia et al., 2024a; Dong & Ma, 2025) with automated conjecturing and proving systems.

Open-endedness and diversity in synthetic data generation. FORMAL DISCO descends from classical open-ended discovery systems like AM and Eurisko (Lenat & Brown, 1984), a lineage recently revived by LLM-based open-ended agents and generators (Wang et al., 2023). In synthetic data generation more broadly, a recurring lesson is that generation must be *seeded* with external sources of randomness to avoid collapsing onto few themes: for instance, TinyStories conditions story generation on random word combinations (Eldan & Li, 2023) and phi injects randomness into prompts for generating synthetic textbooks (Li et al., 2023). Our READMEs and documentation snippets play this same role for verified programs. But where prior work treats diversity as a qualitative design goal, our entropy-maximization framework turns it into a metric that can be tracked and directly optimized.

3 FORMAL DISCO

In this work, our main goal is to enable synthetic generation of programs in languages where a formal verifier is available but potentially few human-written programs exist. Besides boosting LLMs' ability to handle current data-scarce languages, like Dafny, Verus and Frama-C, the system we introduce allows us to picture how languages of the future, which will start with few human-written programs, can be released together with large-scale (synthetic) datasets and benefit from AI assistance.

3.1 A PRINCIPLE OF MAXIMUM ENTROPY FOR SYNTHETIC PROGRAM GENERATION

What makes a good dataset of programs? Several intuitions have been used by prior work on synthetic program generation in Verus — notably in the SAFE and VeruSyn synthetic datasets — as a guide. For one, larger datasets tend to be more useful for training LLMs. But size alone is of course a poor metric: for instance, creating trivial variations of a small set of programs can create size but not useful training signal. Other notions present in existing pipelines include that datasets should cover a variety of language features and programs of varying complexity — both of these served as motivations for VeruSyn's design choices, for instance, to improve over SAFE (Di et al., 2026).

We propose that many intuitively desirable properties — including dataset size, coverage over language features, and various notions of program complexity — can be achieved by a unified goal of seeking *entropy over program features*. More precisely, let $\mathcal{P}$ be the set of valid programs in a given language. A (discrete) program feature is then a function $f : \mathcal{P} \to \mathcal{V}_f$, where $\mathcal{V}_f$ is an enumerable

set of possible values that $f(p)$ can take. Given a finite dataset $\mathcal{D} \in 2^{\mathcal{P}}$, feature f gives rise to a probability distribution $P_f^{\mathcal{D}}(v)$ over the value of $f(p)$ for a program p drawn uniformly from $\mathcal{D}$, where $P_f^{\mathcal{D}}(v) = \frac{1}{|\mathcal{D}|} \sum_{p \in \mathcal{D}} \mathbb{1}[f(p) = v]$. We then define the entropy $H(f, \mathcal{D})$ of feature f in a given dataset $\mathcal{D}$ to be the Shannon entropy of $P_f^{\mathcal{D}}$: $H(f, \mathcal{D}) = -\sum_{v \in \mathcal{V}} P_f^{\mathcal{D}}(v) \log P_f^{\mathcal{D}}(v)$

Intuitively, $H(f, \mathcal{D})$ serves as a measure of the diversity of values of f observed in $\mathcal{D}$. The key principle we build upon is that a *good synthetic dataset of programs will maximize its entropy over a set of relevant program features*. This hypothesis unifies several intuitions on dataset quality that have been implicitly present in recent work on synthetic program generation, including:

Larger datasets can have larger feature entropy. The fact that the uniform distribution has higher entropy than any other probability distribution supported on a discrete set leads to a simple upper bound on feature entropy: $H(f, \mathcal{D}) \leq \log |\mathcal{D}|$. But dataset size alone does not imply usefulness: indeed, an arbitrarily large dataset comprised of copies of the same program will have $H(f, \mathcal{D}) = 0$ for any f, agreeing with the intuition that size is not enough.

High entropy in complexity metrics requires complex programs. One desirable property of $\mathcal{D}$ is that it should contain complex programs. This can also be captured by having high $H(f_c, \mathcal{D})$ over some f_c that is a program feature related to program complexity (e.g., the maximum loop nesting depth in the entire program). If all programs are simple or trivial (e.g., if there are no loops), f_c will collapse into a small number of possible values (e.g., maximum nesting depth in $\{0, 1\}$ if there are no nested loops at all in the corpus). Thus, the absence of complex programs as measured by f_c will be reflected in $\mathcal{D}$ having low $H(f_c, \mathcal{D})$.

Coverage over programming language features can be measured by entropy. The desideratum that a rich dataset should cover the usage of a variety of relevant programming language features (e.g., all of the features appearing in its reference manual (Di et al., 2026)) can also be seen through the lens of entropy maximization. Here, the relevant program features $f_e(p)$ are the binary features checking whether program p uses a given feature e from the target programming language (for instance, whether p uses lemmas, has `#[trigger]` annotations, or ghost code in Rust). If no program in a dataset uses e, the entropy over this feature will be zero: maximizing this entropy will require introducing programs that use e (and others that do not).

Setting $H(f, \mathcal{D})$ as the objective function will serve to inform the design of FORMAL DISCO, the multi-agent system we introduce next. Qualitatively, its workers and components (Section 3.2) will be chosen to introduce entropy over relevant program features f. Quantitatively, Section 3.3 will describe how we directly use dataset entropy as an objective for self-improvement.

3.2 OPEN-ENDED VERIFIED PROGRAM GENERATION WITH LLM-BASED WORKERS

We now describe FORMAL DISCO, a multi-agent system for synthetic program generation. FORMAL DISCO's architecture is inspired by classical *discovery systems* such as AM and Eurisko Lenat & Brown (1984), which aimed at automating mathematical discovery. Here, these "discoveries" will consist of formally verified programs, although the base system can in principle be adapted to other settings including mathematics. The whole system is illustrated in Figure 1. FORMAL DISCO is centered around an "agenda", which keeps track of both tasks and objects. Tasks can point to object dependencies and have a type and other data about the work they require. Furthermore, FORMAL DISCO's agenda is distributed: the agenda exposes a remote API so various *workers* can asynchronously create or claim tasks, as well as create and update objects. This facilitates horizontal scaling, as new agents can enter the system at any time. In worker processes, we use a simple round-robin *scheduler* that runs through a list of workers and invokes them in sequence for one unit of work each. Specifically, we implement the following three workers:

One *run* in FORMAL DISCO consists of starting an agenda and scheduling workers to complete a maximum of K task attempts (up to $30,000$ in our experiments). Open-ended program generation thus starts with the Initiator, which receives a randomly sampled seed context. This context consists of two parts. First, we give the Initiator a sampled GitHub README.md file from a dataset we collected of 100k repositories from GitHub[1]. As we show in the experiments, random READMEs allow us to significantly improve entropy of the generated corpus with respect to program "themes"

[1]These READMEs were sampled from Google's BigQuery GitHub database, and are generally (simply given the overall distribution of programs on GitHub) not coming from verification-related projects.

Table 1: Worker roles that the LLM-based agents in FORMAL DISCO can take.

Worker	Action	Outcome
Initiator	Samples a random seed context and prompts an LLM to generate a small verified program, together with comments suggesting extensions. The program is saved as a new agenda object and checked by the compiler/verifier.	Always creates one new task for the program it wrote: if verification succeeds, creates an `extend` task, otherwise creates a `repair` task.
Fixer	Claims a `repair` task with program and failed compiler/verifier feedback. Prompts an LLM to produce a patch, applies it and reruns the compiler/verifier to get updated feedback.	On success, creates an `extend` task. Otherwise, the repair task remains open with an incremented attempt count. After 3 failed attempts, closes the task.
Extender	Claims an `extend` task for a verified program and prompts an LLM to propose and implement an extension (e.g., new lemma, method, etc) as a patch, which it applies and re-runs the verifier.	If verification succeeds on the new program, creates another `extend` for it, or a `repair` task otherwise.

(e.g., as indicated by the identifiers found in the programs). The second part of the context is a set of up to 2 randomly sampled snippets generated from the language's reference. We prepare these snippets beforehand to contain a description of a feature of the programming language paired with one usage example. This part of the seed helps increase entropy with respect to *language feature coverage*: many features of Dafny or Verus, for instance, do not appear in the generated corpus if seed snippets from the reference are not present.

3.3 DISTILLATION AND SELF-IMPROVEMENT VIA ENTROPY MAXIMIZATION

So far, maximizing entropy has served qualitatively to motivate our system's design. We now go further derive an optimization objective for self-improving our workers using this principle.

In a FORMAL DISCO run, every LLM call is also recorded to the agenda as a *distillation example*, with its full prompt, LLM output, and the *outcome* of the attempt to fulfill the worker's task. We find that existing open-weights LLMs, such as from the Qwen Coder family of models, are generally not yet capable in the verification-aware languages we use. Thus, for each language, we first perform initial runs using closed frontier models (e.g., Claude Opus and Sonnet) to collect initial *distillation* data. Success rate even for these models is in the $22 - 30\%$ range for several worker tasks, making it expensive to exclusively rely on these models to produce a large-scale dataset. However, even a run with $10k$ attempts gives enough distillation data to allow open models to become useful. After this initial seed run, we run subsequent runs using open models. Between runs, we use the data collected so far for supervised fine-tuning (SFT), using LoRA.

As we train, we use *entropy maximization* to formulate an objective to steer workers towards generating increasingly diverse programs, and use a simple ranking and filtering scheme (as in ReST (Gulcehre et al., 2023) and RAFT (Dong et al., 2023)) to optimize for this objective. Specifically, let $f_1, \cdots, f_k$ be a set of program features over which we want the dataset we generate to have high entropy. For each f_i, the definition of $H(f_i, \mathcal{D})$ can be reformulated in terms of contributions of each individual program in $\mathcal{D}$. Specifically, we define the *f-surprisal* $S(f, p, \mathcal{D})$ of program p relative to $\mathcal{D}$ as: $S(f, p, \mathcal{D}) = -\log P_f^{\mathcal{D}}(f(p))$.

Intuitively, the surprisal measures the "rarity" of $f_i(p)$ among programs in $\mathcal{D}$. With S, we can thus rewrite $H(f, \mathcal{D})$ as $H(f, \mathcal{D}) = \frac{1}{|\mathcal{D}|} \sum_{p \in \mathcal{D}} S(f, p, \mathcal{D})$.

This decomposition of $H(f, \mathcal{D})$ in terms of the contributions of each program allows us to rank programs based on the rarity of their feature values. Let $\text{rank}(p, f, \mathcal{D}) \in [1, |\mathcal{D}|]$ be the order of p in $\mathcal{D}$ when programs are sorted by their f-surprisal (with 1 being the program with the highest surprisal value; ties broken arbitrarily). Then, we define the *minimum surprisal rank* (*msr*) of p as: $\textit{msr}(p, \mathcal{D}) = \min_{f \in \{f_1, \cdots, f_k\}} \text{rank}(p, f, \mathcal{D})$.

In each iteration of SFT on the accumulated dataset, we only train on the top $1/3$ successful examples as ranked by the *msr* of the programs they generated, maintaining the fraction of examples of each prompt type available ("initiate", "repair" and "extend"). Our evaluation shows that training with this scheme generates corpora of programs with increasing feature entropy. We also observe qualitative

signals that over time our workers generate richer programs from a verification perspective. For instance, we observe Qwen generating difficult quantifier alternations in its specifications that are never seen in the original Claude data: Claude models never successfully produced specifications requiring more than one $\forall\exists$ quantifier alternation, whereas Qwen in its last iteration produces programs with up to 3 alternations, including one C function with a post-condition with 6 quantifiers and 3 alternations ($\exists\exists\exists\forall\exists\forall$ — used to specify a fairly complex URL validation function).

4 EXPERIMENTS

We now empirically evaluate FORMAL DISCO in two key dimensions: (1) its ability to generate diverse and complex synthetic verified program datasets across multiple languages and (2) the usefulness of the generated datasets for downstream tasks on existing programs. In particular, we answer the following two questions: **RQ1:** Can FORMAL DISCO generate diverse and complex synthetic verified programs at scale across multiple languages? **RQ2:** Are generated programs useful for training models for verification-relevant downstream tasks? Furthermore, we also disentangle, via several ablations, the contributions of the FORMAL DISCO's multi-agent setup as well as the impact of using sampled documentation snippets and GitHub README files on the system's ability to generate diverse datasets. Our dataset is anonymously made available at `https://huggingface.co/datasets/formal-disco-submission/ICLR`.

Languages. We instantiate FORMAL DISCO for three verification-aware languages and tools: Dafny (Leino, 2010), Verus (Lattuada et al., 2023) and Frama-C (Cuoq et al., 2012). All three tools and languages are *auto-active*: regular programs can be augmented by functional specifications (e.g., *requires/ensures* clauses for functions), and proofs are generated automatically by invoking an SMT solver such as Z3 De Moura & Bjørner (2008). Typically, complex specifications cannot be proved fully automatically and require the program to also include helper logical annotations, such as assertions and loop invariants, which can guide the SMT solver (e.g., by decomposing the problem into proving the annotations correct, and then assuming them to prove the verification conditions).

Models. To implement workers in FORMAL DISCO, we use Claude 4.5 Sonnet and Opus (Anthropic, 2025) as closed frontier models, and Qwen 2.5-Coder 32B as an open, dense code generation model (Hui et al., 2024). We use LoRA (Hu et al., 2022) for fine-tuning Qwen 2.5-Coder 32B. Hyperparameters are listed in the appendix.

4.1 EVALUATING SCALABLE SYNTHETIC DATA GENERATION

We first ask whether FORMAL DISCO is effective at generating diverse verified programs at scale; Section 4.3 will then assess whether our synthetic programs help improve models on downstream tasks related to verification. We start by analyzing whether FORMAL DISCO's workers are effective at their own tasks (initiating, fixing and extending programs), and whether the system remains effective and can self-improve when using Qwen2.5-Coder to replace Claude.

Figure 2 shows worker success rates at each of the three FORMAL DISCO generation tasks across Dafny and Verus. Here, the dashed horizontal line indicates worker success rates when Claude models are used for the workers. We find that Claude can initiate a new program successfully between 11% (Verus) and 23% (Dafny) of the time. Repair tends to be a harder task, whereas extending an already working programs tends to be easier, with success rates above 40% in both languages. Importantly, we also find that our pipeline successfully distills and improves this generation capability into Qwen 2.5-Coder: the first iteration of supervised fine-tuning, essentially distilling Claude's responses into Qwen 2.5-Coder, already yields effective workers. After 5 iterations of self-improving, we find that Qwen 2.5-Coder either matches or outperforms the original Claude models across all worker tasks and languages, allowing synthetic data generation to be highly scalable at low cost.

4.2 DIVERSITY

We now evaluate the diversity of the synthetic datasets we generate both within our system and by comparing it to existing datasets (both of human-written and synthetic programs) across languages. Figure 3 shows the feature entropy of the 4 features we optimize for (with the entropy maximization method from Section 3.3): the dashed line again shows the entropy of each feature in the corpus of programs generated from the initial runs with Claude, whereas the solid line shows how this

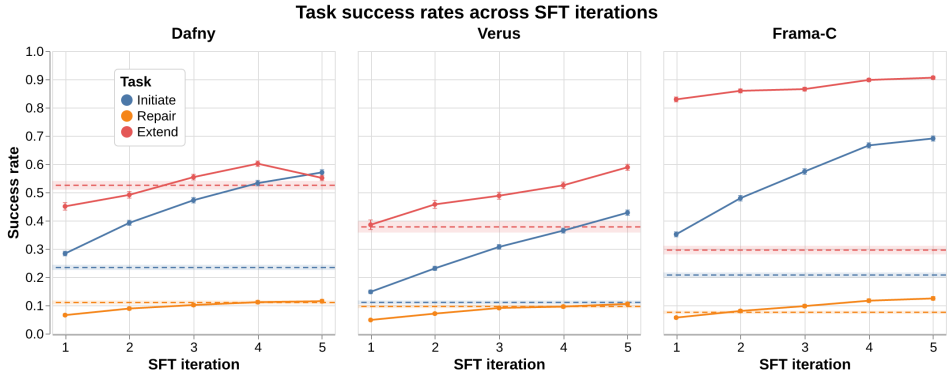

Figure 2: Task success rates for FORMAL DISCO worker tasks. Colors indicate different worker tasks (Initiator, Fixer and Extender). The dashed line shows performance for the seed Claude workers, and solid lines show the Qwen 2.5 Coder 32B model after fine-tuning either on the Claude distillation data (SFT iteration 1) or on the aggregate dataset generated from all previous iterations (SFT iterations ≥ 2). SFT always uses entropy maximization, including on iteration 1. Overall, Qwen's success rate tends to steadily increase, sometimes far surpassing the initial success rate of Claude-based workers.

entropy evolves in datasets generated after each round of SFT with Qwen when applying our entropy maximization method on the combined dataset of all programs generated so far. We observe that, across all features and languages, entropy tends to consistently increase across iterations. In fact, the last generated dataset with Qwen 2.5-Coder is more diverse, as measured by feature entropy, across all 4 features and languages, compared to the initial seed dataset generated by Claude Sonnet and Opus, with the only exception of lemma body sizes in Verus, where entropy increases as we run self-improvement iterations with Qwen but is still behind the Claude-generated datasets by iteration 5. Overall, we observe that FORMAL DISCO workers can effectively *increase diversity* over the chosen-features as they self-improves, in contrast with the conventional wisdom in training LLMs on their own self-generated data where diversity collapse is a common concern.

Qualitative examples of increasing program complexity. Beyond aggregate feature entropy, we also observe a qualitative increase in the complexity of individual programs across self-improvement iterations. Using the size (in lines) of the largest single method or lemma body in a program as a crude proxy for structural complexity, we compare the Claude seed corpus against the last Qwen self-improvement iteration for each language. We observe that median and 90th-percentile sizes (among many other distribution statistics) shift upward significantly in Dafny and Frama-C (e.g., from a median of 13 to 21 lines for Dafny; see Table 4), and likewise the largest verified body produced by the last Qwen iteration exceeds the largest example in the Claude seed corpus by up to 42% across languages (96 vs. 95 lines for Dafny, 105 vs. 86 for Verus, 107 vs. 75 for Frama-C; Table 5). Despite lines of code being a fairly imperfect complexity metric (e.g., it could be exploited by inserting no-op code), manually inspecting the programs shows that they overall contain rather direct attempts at implementing their (increasingly complex) specifications. We show all six of the longest methods/lemmas mentioned above in full in Appendix E.

Ablations. We also ablate the key design choices behind FORMAL DISCO and find that they each contribute measurably to worker effectivenesss and the diversity of the generated corpora. Using a single "monolithic" agent, for instance, decreases success rates and reaches shorter verified programs (e.g., mean 268 with 3 workers vs. 85 lines with a single agent). Removing GitHub READMEs collapses program themes mostly onto textbook data structures and nearly halves worker success rates (12.8% without READMEs vs. 23.0% with, for a monolithic agent). Removing documentation snippets also drastically worsens coverage over language features (e.g., leaves 4 of 30 Dafny language features we extracted from the reference manual entirely unused). More details are in Appendix B.

4.3 DOWNSTREAM TASKS

Finally, we evaluate the performance of models trained on FORMAL DISCO's datasets on verification-related downstream tasks, indirectly reflecting how useful the synthetic data we generate is downstream, beyond intrinsic success and diversity metrics.

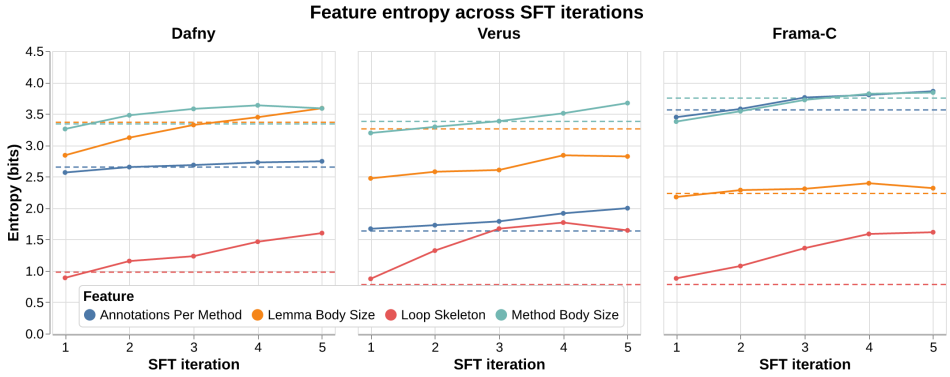

Figure 3: Feature entropy on the program corpora generated by FORMAL DISCO across iterations, on the 4 features that entropy maximization targets (indicated by different colors). The dashed horizontal line shows the feature entropy on the corpus generated in the initial iteration with Claude workers. Then, the solid line shows the entropy of that feature on each run using fine-tuned Qwen-2.5 Coder 32B on the combined corpus of examples generated so far, maximizing entropy using ReST. Overall, FORMAL DISCO effectively improves diversity of the corpus on those features across iterations.

Logical annotation task. We first consider the logical annotation task from prior work (Loughridge et al., 2024; Yang et al., 2025): given a program with formal specifications, insert logical annotations (i.e., assertions, loop invariants) so that the output program passes verification. For this task, we extract SFT training data from FORMAL DISCO's generated programs by simply stripping all annotations from the program and creating one SFT example where the diff that recovers the annotated program is the output, and program and verifier feedback are inputs. We filter out trivial examples (those that verify even after removing annotations) and fine-tune Qwen 2.5-Coder 32B for 15k steps ($<$ 1 epoch on the data) with LoRA on the $program \rightarrow diff$ examples. We do this separately for Dafny and Verus, and evaluate the resulting model on annotating DafnyBench Loughridge et al. (2024) and VerusBench Yang et al. (2025) programs stripped of annotations and again filtering out programs that still verify.

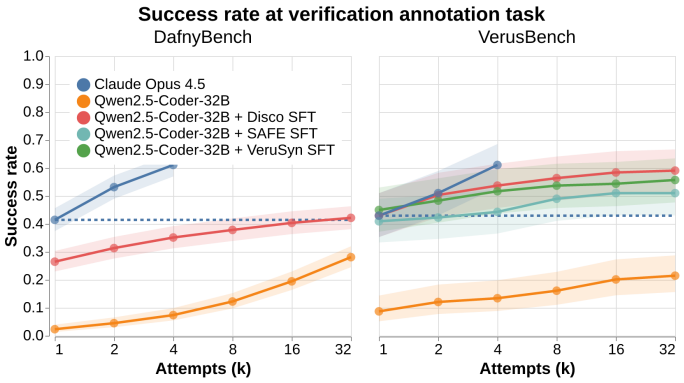

Figure 4: Success rate on the logical annotation task at varying Pass@k, on DafnyBench (left, 555 programs) and VerusBench (right, 149 programs). Bands are 95% boostrapped CIs. The dashed horizontal line shows Pass@1 for Claude 4.5 Opus (blue lines), which we run with up to Pass@4 $k = 4$ attempts. Fine-tuning Qwen 2.5-Coder 32B (red) on our synthetic programs roughly doubles Pass@16 on DafnyBench over the base model (orange) and matches Opus's Pass@1 on VerusBench, where it also outperforms the same model fine-tuned on SAFE (light green) and VeruSyn (dark green) using the same training recipe.

We are unaware of other large-scale datasets for fine-tuning on Dafny; for Verus, we also use our same data extraction pipeline and training recipe using both the SAFE and VeruSyn datasets Chen et al. (2026), which containig 20k and 6.9M synthetic Verus programs respectively. We evaluate Qwen models up to Pass@32, and Claude 4.5 Opus with up to Pass@4 — Claude is prompted to do chain-of-thought reasoning before outputting its diff, whereas our SFT'd models are not yet trained to generate CoTs, given the simple pipeline that extracts training data deterministically from programs. Thus, our results with SFT should be seen as lower bounds on the performance that can be extracted from these datasets.

Figure 4 shows results for both languages in the verification annotation task. On DafnyBench, our fine-tuned Qwen 2.5-Coder at Pass@16 matches the single-shot performance of Claude 4.5 Opus (40.4% vs 41.4%); at this budget, the model trained on FORMAL DISCO data is over 2x more successful than the base Qwen model (at 40.4% *vs* 19.5% Pass@16), showing significant gains from fine-tuning on our synthetic programs. On VerusBench, the Pass@1 performance of fine-tuned Qwen 2.5-Coder matches that of Opus 4.5, at exactly 43%, significantly ahead of the base Qwen 2.5-Coder 32B (8.7% success rate Pass@1). On VerusBench, a further point of comparison for the data quality obtained by FORMAL DISCO is given by using our exact same SFT pipeline using the programs from SAFE, derived from real-world code, and VeruSyn, seeded with SAFE: the SAFE-trained model is comparable at Pass@1 (40.9% vs 43.0% for FORMAL DISCO data), but lags behind with more attempts, plateauing at 51% from Pass@16 onward while the FORMAL DISCO model reaches 59.1% at Pass@32. VeruSyn also leads to similar initial performance, but performs slightly lower at Pass@32 at 54.5% success rate. This suggests that our synthetic data yields more diverse and representative repairs for training.

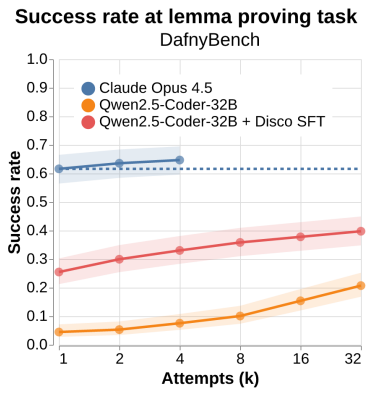

Figure 5: Success rate (Pass@k) on the lemma proving task on lemmas extracted from DafnyBench. Qwen 2.5-Coder 32B fine-tuned on FORMAL DISCO improves significantly on top of the base model, while Opus still performs best.

We furthermore consider the task of proving lemmas: given a program containing a lemma whose body (i.e., its proof) is empty, generate a proof that the verifier accepts. As before, we extract SFT training data directly from FORMAL DISCO's verified programs: we hollow out the body of each lemma and create one SFT example that recovers the original proof, filtering out trivial cases where the hollowed program still verifies (i.e., where Dafny needs no manual proof). We fine-tune Qwen 2.5-Coder 32B on these *program* $\rightarrow$ *proof* examples with the same setup as in the annotation task, and evaluate on DafnyBench lemmas, hollowed and filtered in the same way (357 lemmas in total); we only evaluate on Dafny here, since VerusBench contains no lemmas. Figure 5 shows the results. Fine-tuning on FORMAL DISCO data improves Pass@1 over the base model by more than 5$\times$ (25.5% vs 4.5%); in fact, a single sample from the fine-tuned model outperforms 32 samples from base Qwen 2.5-Coder (20.7% Pass@32), and the gap remains large at every sampling budget (39.8% vs 20.7% at Pass@32). Claude 4.5 Opus remains stronger on this task (61.6% Pass@1), consistent with proof generation benefiting from the chain-of-thought reasoning that our fine-tuned models were not trained to produce. Notably, however, our fine-tuned model continues to improve steadily up to Pass@32, with Opus benefiting less from extra attempts.

Overall, these evaluations show that the fully synthetic programs generated by FORMAL DISCO are highly useful for improving open models on verification tasks, matching a frontier model on Verus annotation and yielding large gains across both languages. Moreover, since SFT on deterministically extracted examples is arguably the simplest possible way to use this data, our results should be read as lower bounds: richer uses of the corpus — e.g., distilling reasoning traces or reinforcement learning against the verifier — are natural directions for future work to make better use of our synthetic data.

5 CONCLUSION

We present FORMAL DISCO, an open-ended system for generating synthetic verified programs at scale. Our results suggest that the long-standing data barrier in formal verification is surmountable: verifiers can provide the scalable quality signal that synthetic data generation needs, and diversity can become a directly optimizable objective as shown by our entropy maximization principle. As AI-generated code becomes ubiquitous, we believe that even future verification-aware languages need not wait for mainstream adoption to accumulate the large-scale corpora that make AI assistance and automation effective — instead, they could launch together with large synthetic datasets, so that even their earliest adopters can program with capable AI support. In an era of increasing reliance on AI-generated code, this approach can help turn strong correctness guarantees a practical default: after all, the bar for software quality should be higher than ever given increasingly powerful tools we have.

AI Disclosure Statement

In this work, we used generative AI tools for assisting with code implementation, proof-reading of the paper, data visualization and analysis. We have used LLMs for generating the synthetic datasets we release, using the reproducible methods described in the paper and the code we will release (rather than using agents in an ad hoc fashion to generate data, for instance). We have not used generative AI tools for brainstorming, running experiments on our behalf, suggesting experiments or hypotheses, or other higher-level research tasks. We have reviewed all AI-assisted work: core abstractions in the code were human-written first before AI assisted with implementing and extending them, and we have reviewed and tested all AI-generated code without AI assistance during reviews (this was often done via pull requests to another author on Github). We take full responsibility for the final content of this work, including text and artifacts produced with the aid of generative AI.

References

Anthropic. Claude Opus 4.5 system card. Technical report, Anthropic, November 2025. URL https://www-cdn.anthropic.com/bf10f64990cfda0ba858290be7b8cc6317685f47.pdf.

Mantas Baksys, Stefan Zetzsche, Olivier Bouissou, Remi Delmas, Soonho Kong, and Sean B. Holden. ATLAS: Automated toolkit for large-scale verified code synthesis, 2026. URL https://arxiv.org/abs/2512.10173.

Tianyu Chen, Shuai Lu, Shan Lu, Yeyun Gong, Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Hao Yu, Nan Duan, Peng Cheng, Fan Yang, Shuvendu K Lahiri, Tao Xie, and Lidong Zhou. Automated proof generation for rust code via self-evolution, 2026. URL https://arxiv.org/abs/2410.15756.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A software analysis perspective. In *International conference on software engineering and formal methods*, pp. 233–247. Springer, 2012.

Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pp. 337–340. Springer, 2008.

Nongyu Di, Tianyu Chen, Shan Lu, Shuai Lu, Yeyun Gong, Peng Cheng, Jacob R. Lorch, Yuan Yao, and Xiaoxing Ma. Reducing the costs of proof synthesis on rust systems by scaling up a seed training set, 2026. URL https://arxiv.org/abs/2602.04910.

Hanze Dong, Wei Xiong, Deepanshu Goyal, Yihan Zhang, Winnie Chow, Rui Pan, Shizhe Diao, Jipeng Zhang, Kashun Shum, and Tong Zhang. Raft: Reward ranked finetuning for generative foundation model alignment. *arXiv preprint arXiv:2304.06767*, 2023.

Kefan Dong and Tengyu Ma. Stp: Self-play llm theorem provers with iterative conjecturing and proving. *arXiv preprint arXiv:2502.00212*, 2025.

Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*, 2023.

Caglar Gulcehre, Tom Le Paine, Srivatsan Srinivasan, Ksenia Konyushkova, Lotte Weerts, Abhishek Sharma, Aditya Siddhant, Alex Ahern, Miaosen Wang, Chenjie Gu, et al. Reinforced self-training (rest) for language modeling. *arXiv preprint arXiv:2308.08998*, 2023.

Nitin Gupta, Shashank Mujumdar, Hima Patel, Satoshi Masuda, Naveen Panwar, Sambaran Bandyopadhyay, Sameep Mehta, Shanmukha Guttula, Shazia Afzal, Ruhi Sharma Mittal, et al. Data quality for machine learning tasks. In *Proceedings of the 27th ACM SIGKDD conference on knowledge discovery & data mining*, pp. 4040–4041, 2021.

David Herel and Tomas Mikolov. Collapse of self-trained language models. *arXiv preprint arXiv:2404.02305*, 2024.

Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.

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

Andrea Lattuada, Travis Hance, Chanhee Cho, Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel. Verus: Verifying rust programs using linear ghost types. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA1):286–315, 2023.

Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, et al. Verus: A practical foundation for systems verification. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pp. 438–454, 2024.

K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International conference on logic for programming artificial intelligence and reasoning*, pp. 348–370. Springer, 2010.

Douglas B. Lenat and John Seely Brown. Why am and eurisko appear to work. *Artificial Intelligence*, 23(3):269–294, 1984. ISSN 0004-3702. doi: https://doi.org/10.1016/0004-3702(84)90016-X. URL https://www.sciencedirect.com/science/article/pii/000437028490016X.

Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need ii: phi-1.5 technical report. *arXiv preprint arXiv:2309.05463*, 2023.

Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, et al. Goedel-prover: A frontier model for open-source automated theorem proving. *arXiv preprint arXiv:2502.07640*, 2025.

Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for formal software verification. *arXiv preprint arXiv:2406.08467*, 2024.

Md Rakib Hossain Misu, Cristina V. Lopes, Iris Ma, and James Noble. Towards AI-assisted synthesis of verified Dafny methods. *Proceedings of the ACM on Software Engineering*, 1(FSE):812–835, July 2024. ISSN 2994-970X. doi: 10.1145/3643763. URL http://dx.doi.org/10.1145/3643763.

Eric Mugnier, Yuanyuan Zhou, Ranjit Jhala, and Michael Coblenz. On the impact of formal verification on software development. *Proceedings of the ACM on Programming Languages*, 9 (OOPSLA2):3642–3668, 2025.

Gabriel Poesia, David Broman, Nick Haber, and Noah D Goodman. Learning formal mathematics from intrinsic motivation. *Advances in Neural Information Processing Systems*, 37:43032–43057, 2024a.

Gabriel Poesia, Chloe Loughridge, and Nada Amin. dafny-annotator: AI-assisted verification of Dafny programs, 2024b. URL https://arxiv.org/abs/2411.15143.

David M Raup. Taxonomic diversity estimation using rarefaction. *Paleobiology*, 1(4):333–342, 1975.

Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. Ai models collapse when trained on recursively generated data. *Nature*, 631(8022):755–759, 2024.

Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *AI Verification: First International Symposium, SAIV 2024, Montreal, QC, Canada, July 22–23, 2024, Proceedings*, pp. 134–155, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-65111-3. doi: 10.1007/978-3-031-65112-0_7. URL https://doi.org/10.1007/978-3-031-65112-0_7.

Trieu H Trinh, Yuhuai Wu, Quoc V Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625(7995):476–482, 2024.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

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

Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R Lorch, Shuai Lu, et al. Autoverus: Automated proof generation for rust code. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA2): 3454–3482, 2025.

A LIMITATIONS

Our work has several limitations that suggest directions for future work. First, our downstream results likely understate the value of the generated data: we train with the simplest possible recipe — supervised fine-tuning on examples extracted deterministically from programs, with no chain-of-thought — and the remaining gap to frontier models on reasoning-heavy tasks such as lemma proving suggests substantial room for richer uses of the corpora, such as distilling reasoning traces or reinforcement learning with verifier feedback. Second, entropy maximization currently targets a small, hand-chosen set of program features; on features we do not optimize for, such as the structure of logical annotations, human-written programs remain more diverse than our synthetic corpora, and automatically discovering *which* features to maximize entropy over (with the meta-level objective of improving downstream utility) is an open problem. Third, open-ended generation still relies on external sources of entropy — random READMEs and documentation snippets — without which we observe thematic collapse; understanding how to sustain open-endedness with less external scaffolding remains a challenge. Finally, while we instantiated FORMAL DISCO for three auto-active verification languages, the framework itself is language-agnostic, and extending it to other formal settings — including interactive theorem provers and mathematics — is a natural next step.

B ABLATIONS

Here, we disentangle the contributions of three design choices in FORMAL DISCO: the division of labor into three specialized workers (as opposed to a single monolithic agent), the GitHub READMEs in the Initiator's seed context, and the documentation snippets sampled from the language reference. Each ablation removes one component and compares the resulting runs on yield (verified programs per LLM call, with a maximum possible of 1 if all calls were successful in generating a verified program) and on the diversity dimension that the component was designed to contribute. Since ablated runs produce corpora of different sizes, all diversity comparisons use rarefaction curves, which compare corpora at matched sample sizes.

Monolithic agent vs. specialized workers. We compare FORMAL DISCO's three workers against a single "monolithic" agent that must produce a complete verified program in a single context window, running both with Claude 4.5 Opus on Dafny. If the generation fails, the monolithic agent is given verified feedback (like the fixer agent) in a growing context and allowed up to 3 attempts in repairing it. The three-worker setup produces verified programs at a higher rate per LLM call (29.8% vs 23.0%), but the more notable difference is in the *complexity* of what each setup can reach: programs from the monolithic agent average 85 lines of code and never exceed 128, whereas the repair-and-extend loop yields programs averaging 268 lines, with the top decile above 598 lines and the largest at over 3,000 lines — verified programs of a size that one-shot generation never attains. This validates the design intuition from Section 3.2: the Initiator can afford to be unambitious because complexity is added incrementally by the Extender, with the Fixer recovering programs that each step breaks.

GitHub READMEs. We next remove the random GitHub READMEs from the seed context, letting the model invent program ideas on its own. Figure 10 (left) shows rarefaction curves of the entropy

over *subject words* (content words extracted from identifiers, our proxy for program themes): without READMEs, theme entropy drops by over 2 bits at every sample size (5.9 vs 8.1 bits at matched $N = 128$; 151 vs 555 unique subject words). The collapse is visible to the naked eye: the most frequent subject words in the no-README corpus are *stack, reverse, insert, pop, push, peek* and *bst* — the model gravitates to a handful of textbook data structures — while the README-seeded corpus is spread over external themes (its most frequent subject word, *cookie*, accounts for only 3.5% of occurrences). Notably, READMEs also nearly *double* the fraction of attempts yielding a verified program (23.0% vs 12.8%): concrete external themes appear to be easier to implement than self-invented ones, making READMEs beneficial even on raw yield.

Documentation snippets. Last, we remove the documentation snippets from the seed context. Here the effect concentrates in the *tail* of the language-feature distribution, which bulk entropy under-reports; Figure 10 (right) therefore shows coverage rarefaction curves for Dafny, counting the number of distinct language features (of the 30 we track, derived from the language's reference) observed as the corpus grows. Corpora generated with documentation snippets quickly cover all $30/30$ features, while no-docs corpora plateau at $26/30$: *bitvectors*, *extreme predicates*, *failure-compatible types* and *iterators/coinduction* never appear at all without documentation snippets — thus, we get zero entropy over those particular features. In entropy terms, both ablations confirm the hypotheses from Section 3.2: READMEs and documentation snippets are complementary external entropy sources — the former over program themes, the latter over language feature usage — and removing either produces a measurable collapse precisely on the corresponding features.

C Comparison to existing datasets via entropy rarefaction curves

We also compare diversity across a range of program features with existing datasets. Here, a major challenge in this comparison is that datasets can vary wildly in size: Verus-Bench, for instance, has less than 200 programs, whereas VeruSyn Di et al. (2026) has over 1 million programs. To perform this analysis, we borrow the conceptual tool of *rarefaction curves*, a commonly used tool in ecology to estimate the taxonomic diversity of a sample of species collected in a given region Raup (1975). A rarefaction curve displays a sample diversity metric in the y axis (e.g., number of distinct species observed, or entropy over label distributions) evolving as we grow our sample size on the x axis. In particular, it allows us to visually compare differently sized datasets by showing how fast diversity grows as we sample from each underlying distribution.

Figure **??** shows rarefaction curves across 4 features, including three that we *do not apply entropy maximization to* (language feature usage, annotation templates, and subject words) and one that we do (method loop skeletons). We make the following observations in each language:

Dafny. The human-written DafnyBench dataset has the most diverse logical annotations, whereas synthetic FORMAL DISCO programs are more diverse on the other 3 features. Even though DafnyBench is considerably smaller than our synthetic datasets, the rarefaction curves over feature entropies over the syntactical skeleton of its logical annotations (e.g., assertions, loop invariants, requires/ensures clauses) shows that DafnyBench's human-written programs are still more diverse over these features. On this feature, which we do not optimize for using entropy maximization, Claude also remains more diverse than Qwen 2.5-Coder even after self-improvement (here we only show the last iteration of self-improvement for clarity; rarefaction curves comparing self-improvement iterations are shown later). In contrast, on the other 3 features (words in identifiers, loop skeletons and usage of Dafny features), we find that FORMAL DISCO-generated programs are *more diverse than DafnyBench*, even when matched for the same sample size.

Verus. In Verus, we compare FORMAL DISCO's corpora with several existing datasets, including two synthetically-generated datasets from prior work (SAFE in cyan and VeruSyn in yellow). Here, we see that rarefaction curves of FORMAL DISCO-generated datasets appear on top in 3 of the 4 features up to their respective lengths (annotation templates, language features, and subject words). In annotation templates, VeruSyn has not exhausted its diversity potential, and entropy keeps increasing (until eventually surpassing FORMAL DISCO's with Claude) even after sampling the entire dataset. In contrast, it saturates much earlier for the other three features. For loop skeletons, the most diverse dataset is VeruSAGE (green), consisting of Verus programs extracted from real-world repositories; here, SAFE also compares favorably to other datasets (despite, for instance, being used as the seed data in VeruSyn, which has less variation in loop structures). Overall, datasets show varied strengths

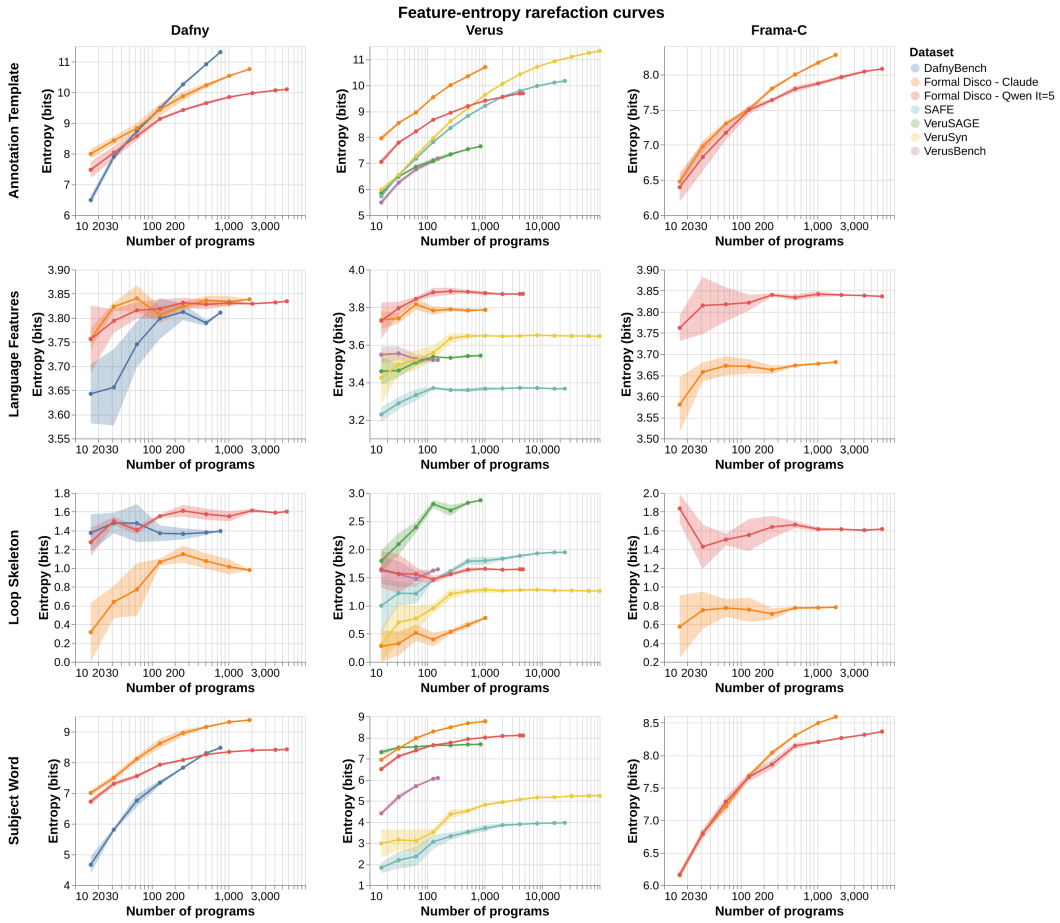

Figure 6: Rarefaction curves over 4 features, including three that we do not optimize for in our entropy maximization procedure (Language Features, Annotation Templates, Subject Words) and one that is optimized for (Loop Skeletons).

in terms of their diversity, with synthetic datasets coming on top of real-world code across some features. Besides showing the competitiveness of FORMAL DISCO-generated corpora, this analysis also illustrates rarefaction curves as a rich tool for comparing the distributions of heterogeneous datasets.

D HYPERPARAMETERS

We report the hyperparameters used for (1) synthetic data generation runs, (2) LoRA fine-tuning during self-improvement and for the downstream tasks, and (3) evaluation.

Generation runs. Each run schedules three workers (Initiator, Fixer, Extender) in round robin. Table 2 lists the settings shared across languages (Dafny, Verus, Frama-C). Runs with Qwen 2.5-Coder models took approximately $8 - 12h$ on a machine with $4x$ NVIDIA H100 GPUs (80 GB VRAM), served to the workers via vLLM. Workers were implemented using LangChain, which allowed accessing both Claude models (on AWS Bedrock) and the vLLM server (serving with OpenAI-compatible API) in a provider-agnostic manner. Each run used 48 parallel worker processes, all of them running the three workers in round-robin.

Sampling temperature and top-p for generation calls were left at each provider's default, for both frontier and open-weights models served via vLLM.

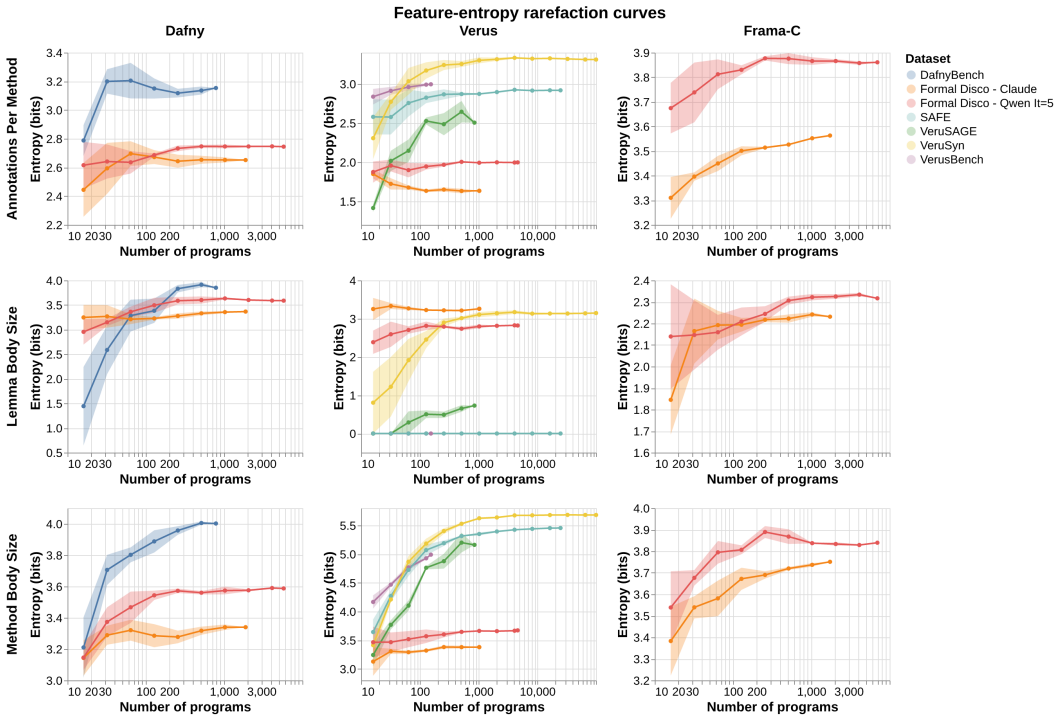

Figure 7: Rarefaction curves across (1) Number of logical annotations per method, (2) Proof/Body length of lemmas, and (3) Body length of methods (both in lines of code). All three are features included in entropy maximization

Table 2: Generation run hyperparameters.

Parameter	Value
Task attempts per run (K)	10k with Claude models, 30k with Qwen
Reference doc snippets sampled per Initiator call	Uniform between 0–2
Max repair attempts before a task is marked failed	3
LLM max output tokens	10,000
Seed frontier models	Claude Opus 4.5, Claude Sonnet 4.5
Self-improvement backbone model	Qwen2.5-Coder-32B-Instruct

LoRA fine-tuning. We use LoRA (Hu et al., 2022) throughout, both for self-improvement iterations on Qwen2.5-Coder-32B-Instruct and for the downstream fixer/lemma-proving tasks. Table 3 lists the shared LoRA and optimization hyperparameters.

Evaluation. We evaluate downstream models with pass@k for $k \in \{1, 2, 4, 8, 16, 32\}$. For the Claude Opus 4.5 baseline we cap sampling at $k = 4$ independent attempts (with chain-of-thought reasoning before the diff) given its cost; open and fine-tuned Qwen models are sampled up to $k = 32$. We report pass@16 as the headline comparison point across models in the main text.

E Qualitative Examples of Increasing Program Corpus Complexity

To complement the aggregate feature-entropy results of Section 4.3, we give complete, verified examples showing that self-improvement with entropy maximization pushes programs into a complexity

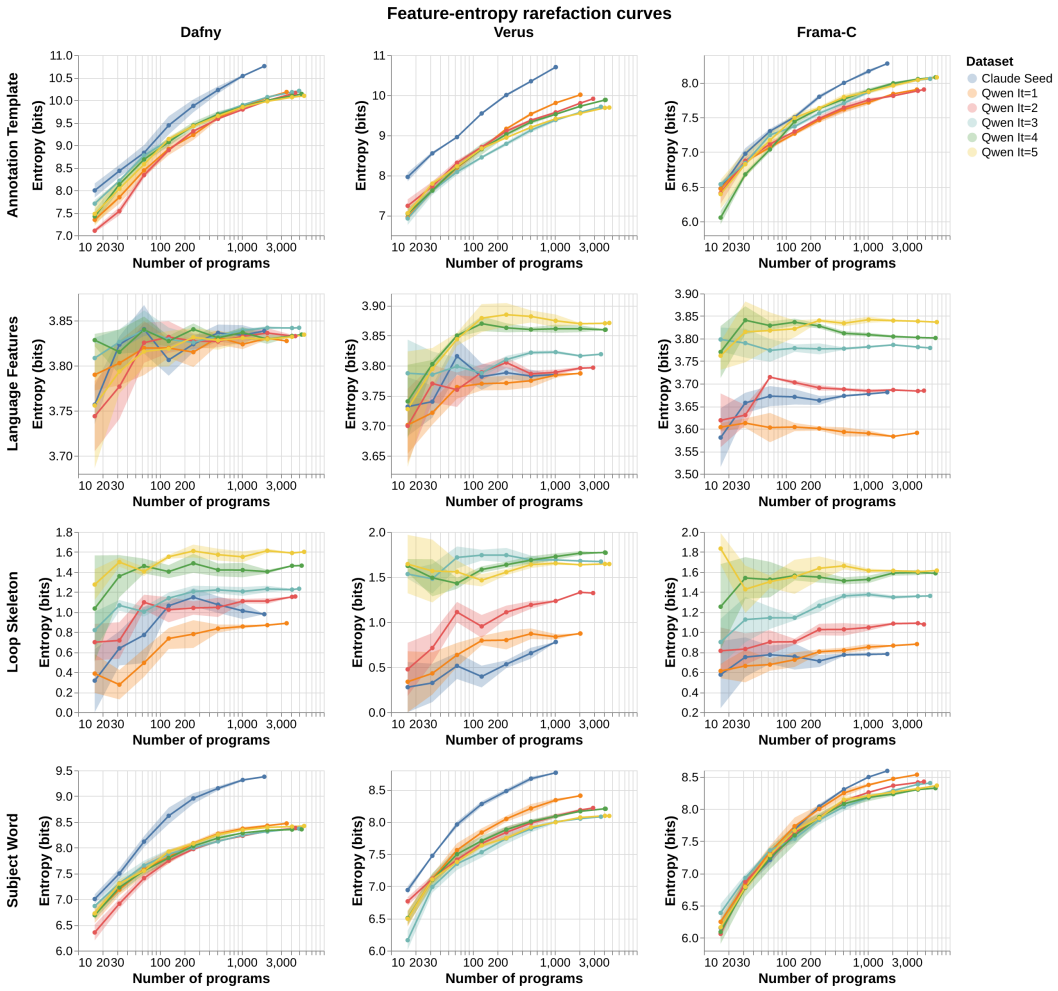

Figure 8: Rarefaction curves across self-improvement iterations, in the same features shown in Figure 6. In features we do not explicitly optimize for, the initial FORMAL DISCO runs with Claude still turn out more diverse; in cases where we do optimize (here, Loop Skeleton), our fine-tuned models take the lead.

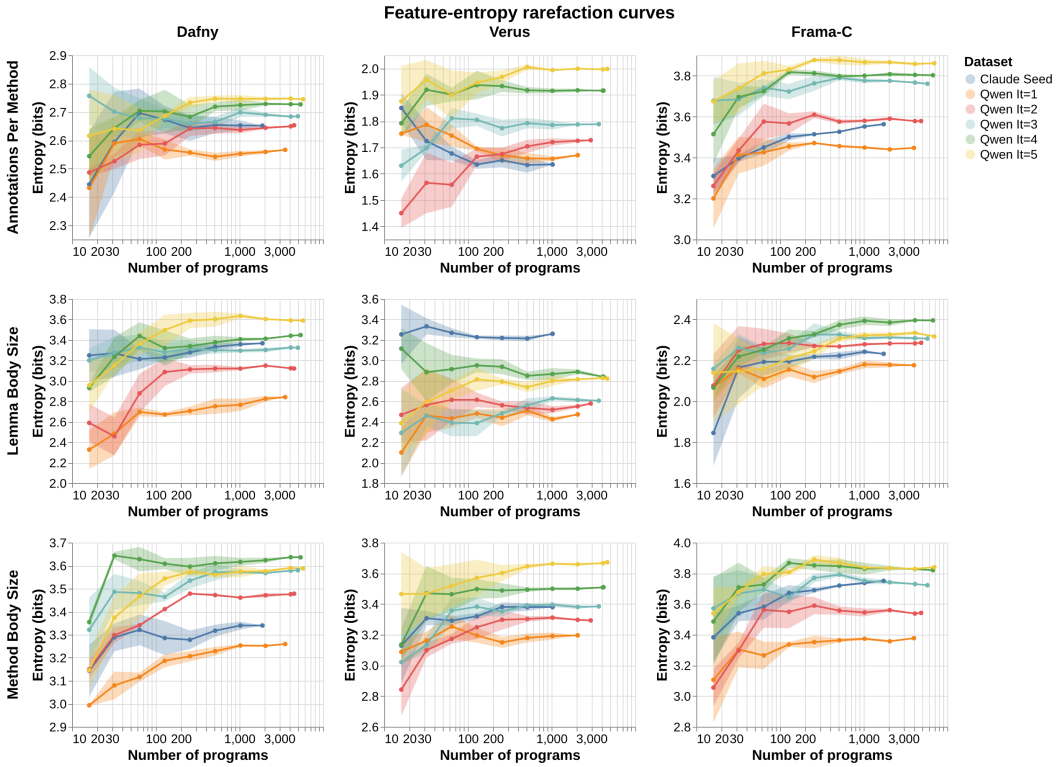

Figure 9: Rarefaction curves across self-improvement iterations, in the same features shown in Figure 7.

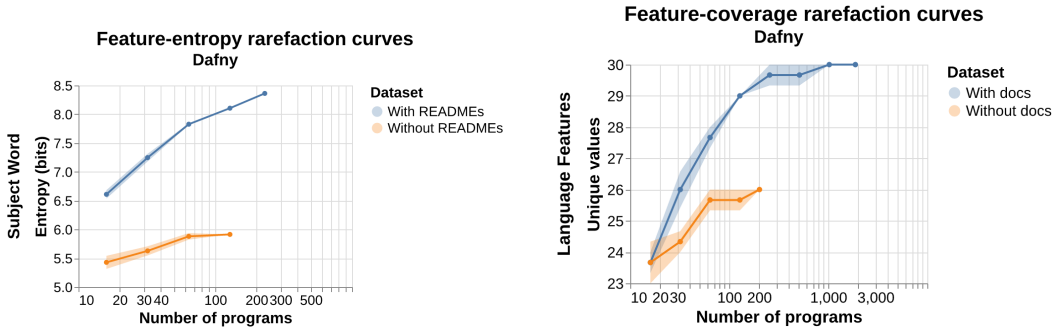

Figure 10: Ablations of the two external entropy sources in the Initiator's seed context, on Dafny. **Left:** entropy over program subject words (themes) with and without GitHub READMEs (monolithic agent, Claude 4.5 Opus); without READMEs, generation highly concentrates onto common textbook data-structure themes and loses in entropy at every corpus size. **Right:** language-feature coverage (number of distinct tracked features observed, of 30) with and without documentation snippets (Claude 4.5 Opus and Sonnet runs pooled per condition); no-docs corpora plateau at 26 of 30 features, with four features never appearing at all. The fact that the orange curves are shorter also shows that the corresponding ablated agent produced less verified programs overall, even with the same LLM call budget.

Table 3: LoRA / SFT hyperparameters.

Parameter	Value
Base model	Qwen2.5-Coder-32B-Instruct
LoRA rank r	32
LoRA α	64
LoRA dropout	0.05
Max sequence length	8192 tokens (longer examples are discarded)
Learning rate	2×10^{-4}
LR scheduler	linear, warmup ratio 0.02
Max gradient norm (gradient clipping)	1.0
Gradient steps	15,000 (< 1 epoch)
Entropy-maximization filtering (self-improvement)	top $1/3$ of successful examples in the accumulated corpus so far, ranked by minimum surprisal rank (mrr), stratified by prompt type (initiate/repair/extend) so that filtering does not change the base proportions of prompt types.

regime the Claude seed corpus never reaches. We use the size (in non-blank lines) of the single largest method or lemma body in a program as a crude scalar proxy for structural complexity — this is directly one of the four features targeted by entropy maximization (Section 3.3). For each language, we compare the Claude Opus/Sonnet seed corpus against the last available self-improvement iteration with Qwen2.5-Coder-32B (Iteration 5 for Dafny and Verus; iteration 2 for Frama-C, which had still completed fewer completed self-improvement rounds at the time of writing).

Table 4: Median and 90th-percentile size (non-blank lines) of the largest method/lemma body per program, Claude seed corpus vs. last Qwen self-improvement iteration. Verus uses the largest *lemma* body rather than method body, since Verus method-body size is dominated by a small number of very long Claude extension chains that are not representative of typical complexity (see text).

	Claude seed		Qwen, last iteration	
Language	median	p90	median	p90
Dafny (method)	13	28	21	41
Verus (lemma)	5	23	3	25
Frama-C (method)	14	25	15	30

Beyond this distributional shift, we manually verified the single largest method or lemma body produced in each corpus. Table 5 shows that on all three languages, the last Qwen iteration produces a verified example substantially past the largest example anywhere in the Claude seed corpus.

Table 5: Longest method or lemma body (non-blank lines) in each synthetic FORMAL DISCO corpus.

Language	Claude seed (max)	Qwen, last iteration (max)
Dafny	95 lines	154 lines
Verus	86 lines	105 lines
Frama-C	75 lines	107 lines

Named declarations, shown in full below: Dafny — `UpdateStatus` vs. `TestPromiseUsage`; Verus — `lemma_insert_maintains_sorted` vs. `values_produce_distinct_chars`; Frama-C — `parse_cookie_pair` vs. `find_cookie_by_index`.

We show each pair below in full.

E.0.1 DAFNY

Dafny — Claude seed corpus | largest verified method (95 lines)

```
method UpdateStatus(recordID: nat, newStatus: TaxonomicStatus, newReference: Option<nat
   >) returns (success: bool)
  requires Valid()
  requires ReferencesValid()
  requires recordID < |records|
  modifies this
  ensures Valid()
  ensures success ==> ReferencesValid()
  ensures !success ==> records == old(records) && ReferencesValid()
  ensures success ==> |records| == old(|records|)
  ensures success ==> records[recordID].status == newStatus
  ensures success ==> records[recordID].acceptedNameUsageID == newReference
  ensures success ==> records[recordID].scientificName == old(records[recordID].
   scientificName)
  ensures success ==> forall i :: 0 <= i < |records| && i != recordID ==> records[i] ==
   old(records[i])
{
  var oldRecord := records[recordID];
  var hasReference := newReference.Some?;

  // Check if the status transition is valid
  if !ValidStatusTransition(oldRecord.status, newStatus, hasReference) {
    success := false;
    return;
  }

  // Additional validation based on new status
  if newStatus == Synonym && newReference.None? {
    // Synonyms must have a reference
    success := false;
    return;
  }

  if newStatus == Accepted && newReference.Some? {
    // Accepted names cannot have a reference
    success := false;
    return;
  }

  // If new status is Synonym, verify the reference points to an Accepted taxon
  if newStatus == Synonym {
    assert newReference.Some?;
    var refID := newReference.value;
    if refID >= |records| || records[refID].status != Accepted || refID == recordID {
      // Cannot create self-referencing synonym or reference non-Accepted taxon
      success := false;
      return;
    }
  }

  // If the old record was Accepted and is being changed to non-Accepted,
  // check if any synonyms reference it
  if oldRecord.status == Accepted && newStatus != Accepted {
    var idx := 0;
    while idx < |records|
      invariant 0 <= idx <= |records|
      invariant forall k :: 0 <= k < idx && records[k].status == Synonym ==>
        records[k].acceptedNameUsageID != Some(recordID)
    {
      if records[idx].status == Synonym && records[idx].acceptedNameUsageID == Some(
   recordID) {
        // Cannot change status of a referenced accepted name
        success := false;
        return;
      }
      idx := idx + 1;
    }
    // At this point, we know no synonym references recordID
    assert forall k :: 0 <= k < |records| && records[k].status == Synonym ==>
      records[k].acceptedNameUsageID != Some(recordID);
  }

  // Create updated record
  var updatedRecord := TaxonRecord(oldRecord.scientificName, newStatus, newReference);
```

```
    // Verify the updated record is valid
    assert ValidRecord(updatedRecord);

    // Update the records sequence
    records := records[recordID := updatedRecord];

    // Verify that other synonym references are still valid
    // Verify that all synonym references are still valid after the update
    ghost var i := 0;
    while i < |records|
      invariant 0 <= i <= |records|
      invariant forall k :: 0 <= k < i && records[k].status == Synonym ==>
        exists j :: 0 <= j < |records| &&
          records[k].acceptedNameUsageID == Some(j) &&
          records[j].status == Accepted
    {
      if records[i].status == Synonym {
        assert records[i].acceptedNameUsageID.Some?;
        var refID := records[i].acceptedNameUsageID.value;
        if i == recordID {
          // This is the record we just updated to Synonym
          // We already verified that refID points to an Accepted taxon
          assert newStatus == Synonym;
          assert newReference.Some?;
          assert refID == newReference.value;
          assert refID < |records|;
          // The record at refID either wasn't changed (if refID != recordID)
          // or it's the new record (if refID == recordID, but that's impossible
          // since we just set recordID to Synonym status)
          // We verified earlier that refID < |records| and records[refID].status ==
     Accepted
          // Since we're updating recordID to Synonym, refID must be different
          if refID == recordID {
            // This would mean we're creating a self-referencing synonym
            // But we checked earlier that records[refID].status == Accepted
            // before the update, and we're changing recordID to Synonym
            assert old(records[recordID].status) == Accepted;
            assert newStatus == Synonym;
            assert false; // Contradiction: we can't reference ourselves
          }
          assert records[refID] == old(records[refID]);
          assert records[refID].status == Accepted;
        } else {
          // This is an old synonym record that wasn't changed
          assert records[i] == old(records[i]);
          assert old(records[i].acceptedNameUsageID) == Some(refID);
          assert old(ReferencesValid());
          assert exists j :: 0 <= j < old(|records|) &&
            old(records[i].acceptedNameUsageID) == Some(j) &&
            old(records[j].status) == Accepted;
          // The refID must have been valid before
          assert refID < |old(records)|;
          assert old(records[refID].status) == Accepted;

          if refID == recordID {
            // This synonym referenced the record we're updating
            // We already checked that if oldRecord was Accepted and we're changing it,
            // no synonyms reference it - so this case is impossible
            assert old(records[recordID].status) == Accepted;
            assert oldRecord.status == Accepted;
            // If we're here and oldRecord.status == Accepted and newStatus != Accepted,
            // then we should have found this earlier and returned false
            if newStatus != Accepted {
              // We checked earlier that no synonyms reference recordID in this case
              assert forall k :: 0 <= k < |records| && records[k].status == Synonym ==>
                records[k].acceptedNameUsageID != Some(recordID);
              assert records[i].status == Synonym;
              assert records[i].acceptedNameUsageID == Some(recordID);
              assert false;
            }
            // We returned false earlier if any synonym referenced recordID
            // If we're here, newStatus == Accepted, so the reference is still valid
            assert newStatus == Accepted;
            assert records[refID].status == Accepted;
          } else {
            // The refID points to a different record that wasn't changed
            assert records[refID] == old(records[refID]);
            assert old(records[refID].status) == Accepted;
```

```
            assert records[refID].status == Accepted;
          }
        }
      }
      i := i + 1;
    }

    success := true;
  }
```

Dafny — Qwen | last self-improvement iteration | largest verified method (154 lines)

```
lemma AncestorsInSetSizeEqualsCount(configLocations: set<Path>, path: Path)
  ensures |AncestorsInSet(configLocations, path)| == CountConfigsInAncestors(
     configLocations, path)
  decreases |path|
{
  if |path| == 0 {
    if path in configLocations {
      assert AncestorsInSet(configLocations, path) == {path};
      assert CountConfigsInAncestors(configLocations, path) == 1;
    } else {
      assert AncestorsInSet(configLocations, path) == {};
      assert CountConfigsInAncestors(configLocations, path) == 0;
    }
  } else {
    AncestorsInSetSizeEqualsCount(configLocations, Parent(path));

    var currentCount := if path in configLocations then 1 else 0;
    var parentCount := CountConfigsInAncestors(configLocations, Parent(path));
    CountConfigsBounded(configLocations, Parent(path));

    // Show the relationship between AncestorsInSet at current level and parent level
    var parentAncestors := AncestorsInSet(configLocations, Parent(path));
var extendedParentAncestors := if path in configLocations then parentAncestors + {path}
     else parentAncestors;

    // Prove the sizes match
    if path in configLocations {
      assert path !in parentAncestors; // path is longer than Parent(path)
      assert extendedParentAncestors == parentAncestors + {path};
      assert |extendedParentAncestors| == |parentAncestors| + 1;

      forall w | w in AncestorsInSet(configLocations, path)
        ensures w in extendedParentAncestors
      {
        assert w in configLocations && IsPrefix(w, path);
        if w == path {
          assert w in extendedParentAncestors;
        } else {
          assert |w| <= |path|;
          assert w == path[..|w|];
          assert Parent(path) == path[..|path|-1];
          assert |w| <= |path| - 1;
          assert w == Parent(path)[..|w|];
          assert IsPrefix(w, Parent(path));
          assert w in parentAncestors;
          assert w in extendedParentAncestors;
        }
      }

      forall w | w in extendedParentAncestors
        ensures w in AncestorsInSet(configLocations, path)
      {
        if w == path {
          assert w in configLocations;
          assert IsPrefix(w, path);
          assert w in AncestorsInSet(configLocations, path);
        } else {
          assert w in parentAncestors;
          assert w in configLocations;
          assert IsPrefix(w, Parent(path));
          // Since |w| <= |Parent(path)| and w == Parent(path)[..|w|]
          assert |w| <= |Parent(path)|;
          assert w == Parent(path)[..|w|];
```

```
          assert IsPrefix(w, path);
          assert IsPrefix(w, Parent(path));
          assert IsPrefix(w, path);
          assert w in AncestorsInSet(configLocations, path);
        }
      }
    } else {
      assert extendedParentAncestors == parentAncestors;

      forall w | w in AncestorsInSet(configLocations, path)
        ensures w in extendedParentAncestors
      {
        assert w in configLocations && IsPrefix(w, path);
        // w is a prefix of path, so either w == path or |w| < |path|
        if w == path {
          assert w in extendedParentAncestors;
        } else {
          assert |w| < |path|;
          assert w == path[..|w|];
          assert Parent(path) == path[..|path|-1];
          assert |w| <= |path| - 1;
          assert w == Parent(path)[..|w|];
          assert IsPrefix(w, Parent(path));
          assert w in parentAncestors;
          assert w in extendedParentAncestors;
        }
        assert IsPrefix(w, Parent(path));
        assert w in parentAncestors;
      }

      forall w | w in extendedParentAncestors
        ensures w in AncestorsInSet(configLocations, path)
      {
        assert w in parentAncestors;
        assert w in configLocations;
        assert IsPrefix(w, Parent(path));
        // Since Parent(path) is a prefix of path and w is a prefix of Parent(path),
        // w is also a prefix of path
        assert |w| <= |Parent(path)|;
        assert w == Parent(path)[..|w|];
        assert Parent(path) == path[..|path|-1];
        assert |w| <= |path| - 1;
        assert w == path[..|w|];
        assert IsPrefix(w, path);
        assert IsPrefix(w, Parent(path));
        assert IsPrefix(w, path);
      }
    }

    assert CountConfigsInAncestors(configLocations, path) == currentCount + parentCount;
    // Help Dafny understand the set equality before asserting cardinality
    assert AncestorsInSet(configLocations, path) == extendedParentAncestors;
    assert |AncestorsInSet(configLocations, path)| == |extendedParentAncestors|;
  }
}
```

E.0.2 VERUS

Verus — Claude seed corpus | largest verified lemma (86 lines)

```
proof fn lemma_insert_maintains_sorted(jobs: Seq<Job>, job: Job)
    requires
        is_sorted_by_priority(jobs),
    ensures
        is_sorted_by_priority(insert_at(jobs, find_insert_position(jobs, job.priority),
     job)),
    decreases jobs.len(),
{
    lemma_insert_position_valid(jobs, job.priority);
    let pos = find_insert_position(jobs, job.priority);
    let result = insert_at(jobs, pos, job);

    if jobs.len() == 0 {
        assert(result.len() == 1);
        assert(is_sorted_by_priority(result));
```

```
    } else if priority_value(job.priority) > priority_value(jobs[0].priority) {
        // Insert at beginning
        assert(pos == 0);
        assert forall|i: int, j: int|
            0 <= i < j < result.len()
        implies
            priority_value(#[trigger] result[i].priority) >= priority_value(#[trigger]
     result[j].priority)
        by {
            if i == 0 {
                // result[0] is job, result[j] is jobs[j-1]
                assert(result[0] == job);
                assert(result[j] == jobs[j - 1]);
                assert(priority_value(job.priority) > priority_value(jobs[0].priority));
                if j > 1 {
                    assert(priority_value(jobs[0].priority) >= priority_value(jobs[j - 1].
     priority));
                }
            } else {
                // Both from original jobs
                assert(result[i] == jobs[i - 1]);
                assert(result[j] == jobs[j - 1]);
            }
        }
    } else {
        // Recursive case
        let rest = jobs.subrange(1, jobs.len() as int);
        lemma_insert_maintains_sorted(rest, job);

// In this branch, priority_value(job.priority) <= priority_value(jobs[0].priority)
// So find_insert_position returns 1 + find_insert_position(rest, ...)
assert(!(priority_value(job.priority) > priority_value(jobs[0].priority)));
// Unfold the definition of find_insert_position for this case
assert(jobs.len() > 0);
assert(jobs.len() != 0);
// The definition of find_insert_position: since jobs.len() > 0 and
// !(priority_value(job.priority) > priority_value(jobs[0].priority)),
// it returns 1 + find_insert_position(rest, job.priority)
assert(find_insert_position(jobs, job.priority) == 1 + find_insert_position(rest, job.
     priority));
let rest_pos = find_insert_position(rest, job.priority);
lemma_insert_position_valid(rest, job.priority);
assert(rest_pos >= 0);
assert(pos == 1 + rest_pos);
assert(pos >= 1);
let rest_result = insert_at(rest, rest_pos, job);

// Prove result[0] == jobs[0] by expanding insert_at definition
assert(result == jobs.subrange(0, pos).push(job).add(jobs.subrange(pos, jobs.len() as int)
     ));
let prefix = jobs.subrange(0, pos);
assert(prefix.len() >= 1);
assert(prefix[0] == jobs[0]);
assert(prefix.push(job)[0] == prefix[0]);
assert(prefix.push(job).add(jobs.subrange(pos, jobs.len() as int))[0] == prefix[0]);

// Show that result == jobs[0] followed by rest_result
        assert forall|i: int, j: int|
            0 <= i < j < result.len()
        implies
            priority_value(#[trigger] result[i].priority) >= priority_value(#[trigger]
     result[j].priority)
        by {
            if i == 0 {
                // result[0] is jobs[0]
                assert(result[0] == jobs[0]);
                if j < pos {
                    // result[j] is jobs[j]
                    assert(result[j] == jobs[j]);
                } else if j == pos {
                    // result[j] is job
                    assert(result[j] == job);
                    assert(priority_value(job.priority) <= priority_value(jobs[0].priority
     ));
                } else {
                    // result[j] is jobs[j-1]
                    assert(result[j] == jobs[j - 1]);
                }
            } else {
```

```
                // Map back to rest_result indices
                let ri = i - 1;
                let rj = j - 1;
                if i < pos && j < pos {
                    assert(result[i] == jobs[i]);
                    assert(result[j] == jobs[j]);
                } else if i < pos && j == pos {
                    assert(result[i] == jobs[i]);
                    assert(result[j] == job);
                    // jobs[i] comes before insertion point, so its priority >= job.
    priority
                    assert(rest_result[ri] == rest[ri]);
                    assert(rest[ri] == jobs[i]);
                } else if i < pos && j > pos {
                    assert(result[i] == jobs[i]);
                    assert(result[j] == jobs[j - 1]);
                } else if i == pos {
                    assert(result[i] == job);
                    assert(result[j] == jobs[j - 1]);
                    // rest_result is sorted, and job is at position rest_pos in
    rest_result
                    assert(rest_result[ri] == job);
                    assert(rest_result[rj] == rest[rj - 1]);
                    assert(is_sorted_by_priority(rest_result));
                } else {
                    // Both after insertion point
                    assert(result[i] == jobs[i - 1]);
                    assert(result[j] == jobs[j - 1]);
                }
            }
        }
    }
}
```

Verus — Qwen | last self-improvement iteration | largest verified lemma (105 lines)

```
proof fn values_produce_distinct_chars(v1: int, v2: int)
    requires
        0 <= v1 < 64,
        0 <= v2 < 64,
        value_to_char_spec(v1) == value_to_char_spec(v2),
    ensures v1 == v2,
{
    // Case analysis based on which ranges v1 and v2 fall into
    // Since value_to_char_spec produces unique characters for each valid value,
    // we can prove that if two values map to the same character, they must be equal.

    // Both in A-Z range
    if v1 < 26 && v2 < 26 {
        let c1 = (('A' as u8 + v1 as u8) as char);
        let c2 = (('A' as u8 + v2 as u8) as char);
        assert(value_to_char_spec(v1) == c1);
        assert(value_to_char_spec(v2) == c2);
        if value_to_char_spec(v1) == value_to_char_spec(v2) {
            assert(c1 == c2);
            assert(('A' as u8 + v1 as u8) as char == ('A' as u8 + v2 as u8) as char);
            assert(v1 as u8 == v2 as u8);
            assert(v1 == v2);
        }
    }
    // Both in a-z range
    else if (v1 >= 26 && v1 < 52) && (v2 >= 26 && v2 < 52) {
        let c1 = (('a' as u8 + (v1 - 26) as u8) as char);
        let c2 = (('a' as u8 + (v2 - 26) as u8) as char);
        assert(value_to_char_spec(v1) == c1);
        assert(value_to_char_spec(v2) == c2);
        if value_to_char_spec(v1) == value_to_char_spec(v2) {
            assert(c1 == c2);
            assert(('a' as u8 + (v1 - 26) as u8) as char == ('a' as u8 + (v2 - 26) as u8)
     as char);
            assert((v1 - 26) as u8 == (v2 - 26) as u8);
            assert(v1 == v2);
        }
    }
    // Both in 0-9 range
```

```
    else if (v1 >= 52 && v1 < 62) && (v2 >= 52 && v2 < 62) {
        let c1 = (('0' as u8 + (v1 - 52) as u8) as char);
        let c2 = (('0' as u8 + (v2 - 52) as u8) as char);
        assert(value_to_char_spec(v1) == c1);
        assert(value_to_char_spec(v2) == c2);
        if value_to_char_spec(v1) == value_to_char_spec(v2) {
            assert(c1 == c2);
            assert((v1 - 52) as u8 == (v2 - 52) as u8);
            assert(v1 == v2);
        }
    }
    // v1 is 62 ('-'), v2 is 62 or 63
    else if v1 == 62 {
        assert(value_to_char_spec(v1) == '-');
        if v2 == 62 {
            assert(value_to_char_spec(v2) == '-');
        } else if v2 == 63 {
            assert(value_to_char_spec(v2) == '_');
        }
    }
    // v1 is 63 ('_'), v2 is 62 or 63
    else if v1 == 63 {
        assert(value_to_char_spec(v1) == '_');
        if v2 == 62 {
            assert(value_to_char_spec(v2) == '-');
        } else if v2 == 63 {
            assert(value_to_char_spec(v2) == '_');
        }
    }
    // v1 in 0-9, v2 in A-Z, a-z, 62, 63
    else if v1 >= 52 && v1 < 62 {
        let c1 = (('0' as u8 + (v1 - 52) as u8) as char);
        assert(value_to_char_spec(v1) == c1);
        // Check v2 in A-Z
        if v2 < 26 {
            let c2 = (('A' as u8 + v2 as u8) as char);
            assert(value_to_char_spec(v2) == c2);
            // Digit != Uppercase letter
        }
        // Check v2 in a-z
        else if v2 < 52 {
            let c2 = (('a' as u8 + (v2 - 26) as u8) as char);
            assert(value_to_char_spec(v2) == c2);
            // Digit != Lowercase letter
        }
        // Check v2 in 62, 63
        else if v2 == 62 {
            assert(value_to_char_spec(v2) == '-');
        }
        else if v2 == 63 {
            assert(value_to_char_spec(v2) == '_');
        }
    }
    // v1 in a-z, v2 in A-Z, 0-9, 62, 63
    else if v1 >= 26 && v1 < 52 {
        let c1 = (('a' as u8 + (v1 - 26) as u8) as char);
        assert(value_to_char_spec(v1) == c1);
        // Check v2 in A-Z
        if v2 < 26 {
            let c2 = (('A' as u8 + v2 as u8) as char);
            assert(value_to_char_spec(v2) == c2);
            // Lowercase != Uppercase
        }
        // Check v2 in 0-9
        else if v2 >= 52 && v2 < 62 {
            let c2 = (('0' as u8 + (v2 - 52) as u8) as char);
            assert(value_to_char_spec(v2) == c2);
            // Lowercase != Digit
        }
        // Check v2 in 62, 63
        else if v2 == 62 {
            assert(value_to_char_spec(v2) == '-');
        }
        else if v2 == 63 {
            assert(value_to_char_spec(v2) == '_');
        }
    }
    // v1 in A-Z, v2 in 0-9, a-z, 62, 63
    else if v1 < 26 {
```

```
        let c1 = (('A' as u8 + v1 as u8) as char);
        assert(value_to_char_spec(v1) == c1);
        // Check v2 in 0-9
        if v2 >= 52 && v2 < 62 {
            let c2 = (('0' as u8 + (v2 - 52) as u8) as char);
            assert(value_to_char_spec(v2) == c2);
            // Uppercase != Digit
        }
        // Check v2 in a-z
        else if v2 >= 26 && v2 < 52 {
            let c2 = (('a' as u8 + (v2 - 26) as u8) as char);
            assert(value_to_char_spec(v2) == c2);
            // Uppercase != Lowercase
        }
        // Check v2 in 62, 63
        else if v2 == 62 {
            assert(value_to_char_spec(v2) == '-');
        }
        else if v2 == 63 {
            assert(value_to_char_spec(v2) == '_');
        }
    }
}
```

E.0.3 FRAMA-C

Frama-C — Claude seed corpus | largest verified function (75 lines)

```
/*@
  requires \valid(cookie_str + (0 .. len-1));
  requires len >= 0 && len < MAX_COOKIE_LEN;
  requires \valid(pair);
  assigns *pair;
  ensures \result == 0 || \result == -1;
  ensures \result == 0 ==> pair->valid == true;
  ensures \result == -1 ==> pair->valid == false;
  ensures pair->key_len < MAX_KEY_LEN;
  ensures pair->value_len < MAX_VALUE_LEN;
*/
int parse_cookie_pair(const char *cookie_str, size_t len, cookie_pair_t *pair) {
    pair->valid = false;
    pair->key_len = 0;
    pair->value_len = 0;


    /*@
      loop invariant 0 <= i <= MAX_KEY_LEN;
      loop assigns i, pair->key[0 .. MAX_KEY_LEN-1];
      loop variant MAX_KEY_LEN - i;
    */
    for (size_t i = 0; i < MAX_KEY_LEN; i++) {
        pair->key[i] = '\0';
    }

    /*@
      loop invariant 0 <= i <= MAX_VALUE_LEN;
      loop assigns i, pair->value[0 .. MAX_VALUE_LEN-1];
      loop variant MAX_VALUE_LEN - i;
    */
    for (size_t i = 0; i < MAX_VALUE_LEN; i++) {
        pair->value[i] = '\0';
    }

    if (len == 0) {
        return -1;
    }


    size_t eq_pos = len;

    /*@
      loop invariant 0 <= i <= len;
      loop invariant eq_pos == len || eq_pos < len;
      loop invariant eq_pos < len ==> cookie_str[eq_pos] == '=';
      loop assigns i, eq_pos;
```

```
      loop variant len - i;
    */
    for (size_t i = 0; i < len; i++) {
        if (cookie_str[i] == '=') {
            eq_pos = i;
            break;
        }
    }


    if (eq_pos == len || eq_pos == 0) {
        return -1;
    }


    size_t key_len = eq_pos;
    if (key_len >= MAX_KEY_LEN) {
        return -1;
    }

    /*@
      loop invariant 0 <= i <= key_len;
      loop invariant i <= MAX_KEY_LEN;
      loop assigns i, pair->key[0 .. MAX_KEY_LEN-1];
      loop variant key_len - i;
    */
    for (size_t i = 0; i < key_len; i++) {
        pair->key[i] = cookie_str[i];
    }
    pair->key[key_len] = '\0';
    pair->key_len = key_len;


    size_t value_start = eq_pos + 1;
    size_t value_len = 0;

    if (value_start < len) {
        value_len = len - value_start;
    }

    if (value_len >= MAX_VALUE_LEN) {
        return -1;
    }

    /*@
      loop invariant 0 <= i <= value_len;
      loop invariant i <= MAX_VALUE_LEN;
      loop assigns i, pair->value[0 .. MAX_VALUE_LEN-1];
      loop variant value_len - i;
    */
    for (size_t i = 0; i < value_len; i++) {
        pair->value[i] = cookie_str[value_start + i];
    }
    pair->value[value_len] = '\0';
    pair->value_len = value_len;

    pair->valid = true;
    return 0;
}
```

Frama-C — Qwen | last self-improvement iteration | largest verified function (107 lines)

```
/*@
  requires \valid_read(cookie + (0 .. cookie_len-1));
  requires \valid(key_out + (0 .. MAX_KEY_LEN-1));
  requires \valid(value_out + (0 .. MAX_VALUE_LEN-1));
  requires cookie_len >= 0;
  requires index >= 0;
  requires \separated(cookie + (0 .. cookie_len-1), key_out + (0 .. MAX_KEY_LEN-1),
     value_out + (0 .. MAX_VALUE_LEN-1));
  assigns  key_out[0 .. MAX_KEY_LEN-1], value_out[0 .. MAX_VALUE_LEN-1];
  ensures  \result == 0 || \result == 1;
*/
int find_cookie_by_index(const char *cookie, size_t cookie_len,
                         size_t index,
```

```
                          char *key_out, char *value_out)
{
  size_t pos = 0;
  size_t current_index = 0;

  /*@
    loop invariant 0 <= pos <= cookie_len;
    loop invariant 0 <= current_index <= index;
    loop assigns   pos, current_index, key_out[0 .. MAX_KEY_LEN-1], value_out[0 ..
     MAX_VALUE_LEN-1];
    loop variant   cookie_len - pos + (index - current_index);
  */
  while (pos < cookie_len) {

    size_t start = pos;

    /*@
      loop invariant start <= pos <= cookie_len;
      loop invariant \forall integer j; start <= j < pos ==> cookie[j] == ' ';
      loop assigns   pos;
      loop variant   cookie_len - pos;
    */
    while (pos < cookie_len && cookie[pos] == ' ') {
      pos++;
    }

    if (pos >= cookie_len || cookie[pos] == ';') {
      break;
    }

    if (current_index == index) {

      size_t key_start = pos;

      /*@
        loop invariant pos <= key_start <= cookie_len;
        loop invariant \forall integer j; pos <= j < key_start ==> cookie[j] != '=';
        loop assigns   key_start;
        loop variant   cookie_len - key_start;
      */
      while (key_start < cookie_len && cookie[key_start] != '=') {
        key_start++;
      }

      if (key_start >= cookie_len) {
        key_out[0] = '\0';
        value_out[0] = '\0';
        return 0;
      }

      size_t key_len = key_start - pos;
      if (key_len >= MAX_KEY_LEN) {
        key_len = MAX_KEY_LEN - 1;
      }

      /*@
        loop invariant 0 <= i <= key_len;
        loop invariant key_len < MAX_KEY_LEN;
        loop assigns   i, key_out[0 .. key_len-1];
        loop variant   key_len - i;
      */
      for (size_t i = 0; i < key_len; i++) {
        key_out[i] = cookie[pos + i];
      }
      key_out[key_len] = '\0';


      size_t value_start = key_start + 1;

      if (value_start < cookie_len) {
        size_t value_end = value_start;

        /*@
          loop invariant value_start <= value_end <= cookie_len;
          loop invariant \forall integer j; value_start <= j < value_end ==>
                         (cookie[j] != ';' && cookie[j] != '\0');
          loop assigns   value_end;
          loop variant   cookie_len - value_end;
        */
```

```
        while (value_end < cookie_len && cookie[value_end] != ';' && cookie[value_end] !=
    '\0') {
          value_end++;
        }

        size_t value_len = value_end - value_start;
        if (value_len >= MAX_VALUE_LEN) {
          value_len = MAX_VALUE_LEN - 1;
        }

        /*@
          loop invariant 0 <= i <= value_len;
          loop invariant value_len < MAX_VALUE_LEN;
          loop assigns   i, value_out[0 .. value_len-1];
          loop variant   value_len - i;
        */
        for (size_t i = 0; i < value_len; i++) {
          value_out[i] = cookie[value_start + i];
        }
        value_out[value_len] = '\0';

        return 1;
      }

      key_out[0] = '\0';
      value_out[0] = '\0';
      return 0;
    }

    current_index++;


    size_t semi_colon = pos;

    /*@
      loop invariant pos <= semi_colon <= cookie_len;
      loop invariant \forall integer j; pos <= j < semi_colon ==>
                     (cookie[j] != ';' && cookie[j] != '\0');
      loop assigns   semi_colon;
      loop variant   cookie_len - semi_colon;
    */
    while (semi_colon < cookie_len && cookie[semi_colon] != ';' && cookie[semi_colon] !=
    '\0') {
      semi_colon++;
    }

    if (semi_colon < cookie_len && cookie[semi_colon] == ';') {
      pos = semi_colon + 1;
    } else {
      pos = semi_colon;
      break;
    }
  }

  key_out[0] = '\0';
  value_out[0] = '\0';
  return 0;
}
```

F Implementation Details

We describe here implementation-level details of Formal Disco that complement the system description in Section 3.2.

Agenda tasks and priorities. The distributed agenda exposes a small custom RPC server with operations for workers to read and write tasks and objects (we also have a local agenda implementation for small / single-process runs). Concurrency in the agenda is controlled by a single lock, held while claiming tasks or creating/updating objects; a worker claims the next available task by walking a priority-sorted list and atomically marking it as being worked on. Objects store a path, type, parent object paths, content, arbitrary properties, and an *interestingness* score; tasks store a type, parent object paths, and mutable status (new/attempted/being-worked-on/done/failed) together with an attempt counter. The effective priority of a task is its base priority multiplied by the interestingness of

every object it depends on, so tasks attached to more "interesting" objects. While we attempted to experiment with this "interestingness" factor, we did not end up relying on it much to influence the dynamics of the system. Thus, although this is currently one knob in the implementation that could be used to focus a run on programs that appeared to have rare or promising features (as opposed to the entropy maximization fine-tuning, which only takes place after a full run),

Checkpointing. The agenda checkpoints itself periodically based on the number of RPC calls processed, serializing its full state (tasks, objects, task outcomes, and attempt counters) atomically to disk. On restart, any task that was marked as being worked on when the process stopped is reset to a retryable state, so an interrupted worker never leaves a task permanently stuck. Agenda checkpoints are the primary output of the system: the final checkpoint is where we take both full verified examples (for downstream fine-tuning) as well as distillation examples (for the 3 worker tasks) used for self-improvement.

Verification pipeline. Each language backend shares a common interface with three possible verification outcomes: SUCCESS, GOALUNPROVEN (the program type-checks and the verifier runs, but some proof obligation is left unproven), and FAIL (syntax or other hard failures). Verification is performed by invoking the relevant compiler/verifier with a timeout, and classifying the outcome from its standard output/error. The verifier's raw stdout and stderr are passed to the LLM verbatim as the "notes" in the "repair" prompts (Appendix H).

Scheduler. The round-robin scheduler cycles through the configured workers indefinitely, giving each one its fixed fuel budget per turn, until the agenda signals that the run's maximum number of task attempts has been reached. A worker with no eligible task simply ends its turn early, leaving any unused fuel unspent; there is no explicit signal exchanged between workers to skip a turn.

Distillation examples. Every LLM call a worker makes is recorded as a distillation example object with four core fields: the prompt type (e.g., `initiate`, `repair`, `extend`), the arguments used to build the prompt (e.g., sampled repository, README, and doc snippets for `initiate`; the failing program and verifier notes for `repair`), the raw model response, and the outcome of the resulting attempt. These examples are the basis for both the initial distillation from frontier models and the entropy-maximization filtering used in self-improvement (Section 3.3).

G PROGRAM FEATURES

Table 6 lists the program features $f : \mathcal{P} \to \mathcal{V}_f$ used throughout our feature-entropy analysis (Section 3.3). The first four features are the ones we directly target for entropy maximization during self-improvement; the last three (language feature usage, annotation templates, and subject words) are used only for comparing datasets in Section 4.3 and are not optimized against.

Table 6: Program features used for feature-entropy analysis. "Optimized" features are the targets of entropy maximization during self-improvement (Section 3.3).

Feature	Optimized?	One-line summary
Annotations per method	Yes	# of logical annotations attached to a method.
Lemma body size	Yes	# of non-blank lines in a lemma's body.
Loop skeleton	Yes	syntactic shape of a method's (possibly nested) loops.
Method body size	Yes	# of non-blank lines in a method's body.
Language feature usage	No	which language constructs a program uses.
Annotation template	No	syntactic skeleton of an annotation, identifiers abstracted.
Subject words	No	lemmatized nouns/verbs from declaration names.

Features are multisets, not scalars, per program. A single program almost always contributes more than one observation to a feature: a program with several methods has one loop skeleton and

one annotation count *per method*, a program with several lemmas has one lemma size *per lemma*, and a program's language-feature usage and identifiers are tallied over every occurrence in the program. Concretely, our implementation computes each feature as a `Counter` over the program (a multiset mapping each observed value to how many times it occurs), rather than as a single scalar. This requires no change to the entropy-maximization framework of Section 3.3: we simply pool every program's counter into a single dataset-level counter before normalizing to obtain $P_f^{\mathcal{D}}$ in Equation 3.1, so a program with more methods, lemmas, or annotations contributes proportionally more mass to the pooled empirical distribution, much like how a longer document contributes more n-grams to a corpus-level language model.

We illustrate all seven features on the same example Dafny program below, showing the exact output of our feature extractor on it.

Example program used throughout this section

```
// Returns the index of the maximum element of a nonempty array.
method FindMaxIndex(values: array<int>) returns (maxIndex: int)
  requires values.Length > 0
  ensures 0 <= maxIndex < values.Length
  ensures forall k :: 0 <= k < values.Length ==> values[k] <= values[maxIndex]
{
  maxIndex := 0;
  var i := 1;
  while i < values.Length
    invariant 1 <= i <= values.Length
    invariant 0 <= maxIndex < i
    invariant forall k :: 0 <= k < i ==> values[k] <= values[maxIndex]
  {
    if values[i] > values[maxIndex] {
      maxIndex := i;
    }
    i := i + 1;
  }
}

// Checks whether an array is sorted in non-decreasing order.
method IsSorted(values: array<int>) returns (result: bool)
  ensures result <==> forall a, b :: 0 <= a <= b < values.Length ==> values[a] <= values[b
    ]
{
  result := true;
  var i := 1;
  while i < values.Length
    invariant 1 <= i <= values.Length
    invariant result <==> forall a, b :: 0 <= a <= b < i ==> values[a] <= values[b]
  {
    if values[i-1] > values[i] {
      result := false;
    }
    i := i + 1;
  }
}

// The maximum index found by FindMaxIndex is the unique such index.
lemma MaxIndexIsUnique(values: array<int>, i: int, j: int)
  requires values.Length > 0
  requires 0 <= i < values.Length && 0 <= j < values.Length
  requires forall k :: 0 <= k < values.Length ==> values[k] <= values[i]
  requires forall k :: 0 <= k < values.Length ==> values[k] <= values[j]
  requires values[i] != values[j]
  ensures false
{
  assert values[i] <= values[j];
  assert values[j] <= values[i];
}
```

Annotations per method. Counts requires, ensures, invariant, assert, and decreases clauses attached to a method (lemmas are excluded, see "Lemma size" below). On the example program, this yields *two* observations, one per method: `FindMaxIndex` has 6 (1 requires + 2 ensures + 3 invariants) and `IsSorted` has 3 (1 ensures + 2 invariants). Extracted multiset: $\{6 \mapsto 1,\ 3 \mapsto 1\}$.

Lemma body size. Non-blank lines in a lemma's body. The example program has a single lemma, `MaxIndexIsUnique`, whose two-line body (the two `assert` statements) gives the multiset $\{2 \mapsto 1\}$.

Loop skeleton. The syntactic shape of a method's loops, with all non-loop code (e.g., the `if` statements guarding the updates) discarded. Both `FindMaxIndex` and `IsSorted` contain a single, non-nested `while` loop with no other loops inside it, so both reduce to the same skeleton and the example yields the multiset $\{$`"while { }"` $\mapsto 2\}$ — illustrating that repeated values are common and are exactly what the entropy of this feature is sensitive to.

Method body size. Non-blank lines in a method's body, excluding its signature and specification clauses. The example program's two methods give the multiset $\{12 \mapsto 1,\ 11 \mapsto 1\}$ (`FindMaxIndex` and `IsSorted`, respectively).

Language feature usage. For each of a curated set of language constructs (one short description with a usage example per construct, the same descriptions sampled as reference snippets into Initiator prompts, Appendix H), a regular expression counts how many times the construct is used in the program, giving one count per detected construct. On the example program, this yields a multiset with 9 distinct constructs, each with its own occurrence count, from a single 40-line program: $\{$`logical-ops` $\mapsto 9$, `lambdas-arrows` $\mapsto 8$, `quantifiers` $\mapsto 6$, `numeric-types` $\mapsto 6$, `arrays` $\mapsto 3$, $\ldots\}$. We curated such constructs from each language's reference manual: 30 for Dafny, 30 for Verus, and 28 for Frama-C.

Annotation template. The syntactic skeleton of an assertion, invariant, requires, or ensures clause, with every identifier-shaped token replaced by $\star$ (a fixed set of language keywords and built-ins, e.g. `forall`, is left unabstracted). Even our small three-declaration example program yields 13 distinct templates (each occurring once or twice), e.g. $\{$`"requires: *.* > 0"` $\mapsto 2$, `"assert: *[*] <= *[*]"` $\mapsto 2$, `"invariant: 0 <= * < *"` $\mapsto 1$, $\ldots\}$ — illustrating how quickly this feature's support grows even on small programs, which is part of why it saturates slowly in the rarefaction curves of Section 4.3.

Subject words. Declaration names are split at `camelCase` boundaries, part of speech-tagged, lemmatized, and filtered to nouns and non-stopword verbs. From the three declaration names in the example program (`FindMaxIndex`, `IsSorted`, `MaxIndexIsUnique`), this yields the multiset $\{$`index` $\mapsto 2$, `find` $\mapsto 1$, `sort` $\mapsto 1\}$: `max` and `unique` are dropped because the part-of-speech tagger does not tag them as a noun or verb in context, and `is` is dropped as a stopword.

H Prompts and Example Generations

We give the exact prompts used by the Initiator worker (Section 3.2) for each language, and show one successful generation from the self-improved Qwen2.5-Coder-32B model for each language, taken from the last self-improvement iteration. Fixer and Extender prompts follow the same diff-based instructions across languages, differing only in the target language's name and verifier; we show the Dafny repair prompt as a representative example.

H.0.1 Initiator prompts

Dafny — Initiator system prompt

```
You are an expert Dafny programmer. You will receive a GitHub repository name and its
     README. Your task is to:
1. Come up with a concise idea for a Dafny program inspired by the repository's theme. The
      repository is most likely unrelated to verified programming, so freely adapt or
     reinterpret its theme.
2. Immediately implement that idea as a self-contained Dafny program.

Your program should NOT try to implement the entire idea, which is likely to be overly
     ambitious to write in one go.
```

```
Start with e.g. a few functions at most, or a very basic class with only a couple of core
     methods, or prove a basic lemma, etc. You can also add comments on ideas to extend
     the program later.
The output must be valid Dafny code and compile/verify when possible. Keep this initial
     program concise.

Output ONLY Dafny source code. Include a brief comment at the top of the program
     describing the idea.
```

Verus — Initiator system prompt

```
You are an expert Verus programmer. You will receive a GitHub repository name and its
     README. Your task is to:
1. Come up with a concise idea for a Verus program inspired by the repository's theme. The
      repository is most likely unrelated to verified programming, so freely adapt or
     reinterpret its theme.
2. Immediately implement that idea as a self-contained Verus program.

Your program should NOT try to implement the entire idea, which is likely to be overly
     ambitious to write in one go.
Start with e.g. a few functions at most, or prove a basic lemma, etc. You can also add
     comments on ideas to extend the program later.
The output must be valid Verus code (Rust with Verus verification annotations) and compile
     /verify when possible. Keep this initial program concise.

Output ONLY Verus source code. Include a brief comment at the top of the program
     describing the idea.
```

Frama-C — Initiator system prompt

```
You are an expert C programmer who writes formally verified code using Frama-C and ACSL.
     You will receive a GitHub repository name and its README. Your task is to:
1. Come up with a concise idea for a C program with ACSL annotations inspired by the
     repository's theme. The repository is most likely unrelated to formal verification,
     so freely adapt or reinterpret its theme.
2. Immediately implement that idea as a self-contained C program with ACSL annotations.

Your program should NOT try to implement the entire idea, which is likely to be overly
     ambitious to write in one go.
Start with e.g. a few functions at most, or prove a basic lemma, etc. You can also add
     comments on ideas to extend the program later.
Include function contracts (requires, ensures, assigns) and loop annotations (loop
     invariant, loop assigns, loop variant) as appropriate.
The output must be valid C code with ACSL annotations and verify with frama-c -wp when
     possible. Keep this initial program concise.

Output ONLY C source code with ACSL annotations. Include a brief comment at the top
     describing the idea.
```

The user message follows a shared template across languages, populated with a sampled repository name, its README, and up to 2 sampled reference snippets (Section 3.2):

Initiator user message template

```
Repository: {repo}

README:
{readme}

[optional: up to 2 sampled reference snippets, see below]
Come up with an idea for a {LANGUAGE} program inspired by this repository and implement it.
      Output only {LANGUAGE} source code.
```

where each reference snippet takes the form:

Reference snippet format

```
Reference snippets for some {LANGUAGE} language constructs you may find useful:

--- {feature_id} ---
{feature description and one usage example, sampled from the language reference}
```

H.0.2 REPAIR PROMPT (REPRESENTATIVE)

Dafny — Fixer system prompt

```
You are an expert Dafny developer. You will be given a Dafny program that has errors
     pointed out by Dafny. These errors can be syntactic, or failures to verify the
     program (i.e. prove post-conditions or verify current assertions/invariants).
Your job is to repair these errors by emitting a DIFF in a simple, line-based format.

Diff format:
- Lines starting with '@@' are anchors (search-forward markers). These don't modify the
     program, but just start a new 'block' of changes in your patch.
- Lines starting with '=' keep that exact line: find it forward and advance the cursor.
     You typically only need a few of these after your @@ line to position the cursor for
     the actual changes: you don't need to copy much of the original file.
- Lines starting with '-' delete that exact line found forward.
- Lines starting with '+' add a new line at the current cursor.

- All diff lines should start with one of the special characters above and a space
     following them. Other lines will be completely ignored
Here is an example of a diff:

Text before:
{example_before}

Example of model output (diff in the format you must follow):
{example_diff}

Text after:
{example_after}
```

H.0.3 EXAMPLE GENERATIONS

We show one successful program generated by the self-improved Qwen2.5-Coder-32B model for each language, together with the sampled README and reference snippet(s) that seeded it. All three examples come from the Initiator task and pass compilation and verification.

Dafny. Sampled repository: `TGITS/programming-workouts`.

Dafny example — sampled README

```
Repository: TGITS/programming-workouts

# Hello World

The classical introductory exercise. Just say "Hello, World!".

["Hello, World!"](http://en.wikipedia.org/wiki/%22Hello,_world!%22_program) is
the traditional first program for beginning programming in a new language
or environment.

The objectives are simple:

- Write a function that returns the string "Hello, World!".
- Run the test suite and make sure that it succeeds.
- Submit your solution and check it at the website.

If everything goes well, you will be ready to fetch your first real exercise.

## Running the tests
```

```
To run the tests, run the command `dotnet test` from within the exercise directory.

## Autoformatting the code

F# source code can be formatted with the [Fantomas](https://github.com/fsprojects/fantomas
    ) tool.

After installing it with `dotnet tool restore`, run `dotnet fantomas .` to format code
    within the current directory.

## Further information

For more detailed information about the F# track, including how to get help if
you're having trouble, please visit the exercism.io [F# language page](http://exercism.io/
    languages/fsharp/resources).

## Source

This is an exercise to introduce users to using Exercism [http://en.wikipedia.org/wiki/%22
    Hello,_world!%22_program](http://en.wikipedia.org/wiki/%22Hello,_world!%22_program)
```

Dafny example — sampled reference snippet

```
--- quantifiers ---
Quantifier expressions `forall x :: P(x)` and `exists x :: P(x)` are first-class boolean
    expressions. Bound variables may be constrained by a range (`forall x | C(x) :: P(x)
    `). Triggers (e.g., `{:trigger}` or matched terms appearing in P) guide the SMT
    solver in instantiating quantifiers.

```dafny
predicate AllPositive(s: seq<int>)
{
  forall i :: 0 <= i < |s| ==> s[i] > 0
}

predicate HasZero(s: seq<int>)
{
  exists i :: 0 <= i < |s| && s[i] == 0
}

lemma PositiveImpliesNoZero(s: seq<int>)
  requires AllPositive(s)
  ensures !HasZero(s)
{
}

lemma BoundedRange(n: int)
  requires n >= 0
  ensures forall k :: 0 <= k <= n ==> 0 <= k && k <= n
{
}

method UseQuantifiers()
{
  var s := [1, 2, 3];
  assert AllPositive(s);
  PositiveImpliesNoZero(s);
  assert !HasZero(s);
}
```
```

Dafny example — Qwen2.5-Coder-32B response (verifies successfully)

```
// Inspired by "programming-workouts" - A classic "Hello, World!" exercise
// implemented in Dafny with verification. We define a function that returns
// "Hello, World!" and prove properties about string concatenation and
// character sequences.

function HelloWorld(): string
{
  "Hello, World!"
```

```
}

// Prove that our HelloWorld function returns a string of expected length
lemma HelloWorldLength()
  ensures |HelloWorld()| == 13
{
}

// String concatenation utility
function Concat(s1: string, s2: string): string
{
  s1 + s2
}

// Prove that concatenation preserves lengths
lemma ConcatLength(s1: string, s2: string)
  ensures |Concat(s1, s2)| == |s1| + |s2|
{
}

// Prove that concatenating empty string doesn't change the string
lemma ConcatEmptyRight(s: string)
  ensures Concat(s, "") == s
{
}

lemma ConcatEmptyLeft(s: string)
  ensures Concat("", s) == s
{
}

// Extract greeting and audience from HelloWorld
lemma HelloWorldStructure()
  ensures HelloWorld() == "Hello" + ", World!"
  ensures |"Hello"| == 5
  ensures |", World!"| == 8
{
}

// A method that uses our HelloWorld function
method SayHello() returns (greeting: string)
  ensures greeting == "Hello, World!"
  ensures |greeting| == 13
{
  greeting := HelloWorld();
  HelloWorldLength();
}

// Future extensions could include:
// - More international greetings with locale parameters
// - ASCII art rendering of "Hello World"
// - Verification of different string operations (substring, replace, etc.)
// - Unicode character properties
```

Verus. Sampled repository: `NirmalL/OWA`.

Verus example — sampled README

````
Repository: NirmalL/OWA

# cookie [![Build Status](https://secure.travis-ci.org/defunctzombie/node-cookie.png?
     branch=master)](http://travis-ci.org/defunctzombie/node-cookie) #

cookie is a basic cookie parser and serializer. It doesn't make assumptions about how you
     are going to deal with your cookies. It basically just provides a way to read and
     write the HTTP cookie headers.

See [RFC6265](http://tools.ietf.org/html/rfc6265) for details about the http header for
     cookies.

## how?

```
npm install cookie
```
````

````
```javascript
var cookie = require('cookie');

var hdr = cookie.serialize('foo', 'bar');
// hdr = 'foo=bar';

var cookies = cookie.parse('foo=bar; cat=meow; dog=ruff');
// cookies = { foo: 'bar', cat: 'meow', dog: 'ruff' };
```

## more

The serialize function takes a third parameter, an object, to set cookie options. See the
    RFC for valid values.

### path
> cookie path

### expires
> absolute expiration date for the cookie (Date object)

### maxAge
> relative max age of the cookie from when the client receives it (seconds)

### domain
> domain for the cookie

### secure
> true or false

### httpOnly
> true or false
````

Verus example — sampled reference snippet

````
--- while-loops ---
A verified `while` loop carries `invariant` clauses that hold before the loop, after every
     iteration, and (with the negated guard) at the loop's exit. A `decreases` clause
    names a value that strictly decreases each iteration, proving termination.

```rust
use vstd::prelude::*;

verus! {

fn count_to(n: u32) -> (r: u32)
    requires n <= 1_000_000,
    ensures r == n,
{
    let mut i: u32 = 0;
    while i < n
        invariant
            i <= n,
            n <= 1_000_000,
        decreases n - i,
    {
        i = i + 1;
    }
    i
}

} // verus!
```
````

Verus example — Qwen2.5-Coder-32B response (verifies successfully)

```
// Inspired by the cookie parsing library: A verified simple key-value pair parser
// for HTTP-style cookies (e.g., "key1=value1; key2=value2").
// We verify properties about parsing and serialization of cookie strings.

use vstd::prelude::*;
```

```
use vstd::string::*;

verus! {

// Represents a parsed cookie as a key-value pair
pub struct Cookie {
    pub key: Vec<u8>,
    pub value: Vec<u8>,
}

impl Cookie {
    pub open spec fn well_formed(&self) -> bool {
        &&& self.key.len() > 0
        &&& self.value.len() > 0
    }

    pub open spec fn key_len(&self) -> nat {
        self.key.len() as nat
    }

    pub open spec fn value_len(&self) -> nat {
        self.value.len() as nat
    }
}

// Spec function to check if a character is valid in a cookie name/value
// (simplified: alphanumeric only for this example)
pub open spec fn is_valid_cookie_char(c: u8) -> bool {
    (c >= 'a' as u8 && c <= 'z' as u8) ||
    (c >= 'A' as u8 && c <= 'Z' as u8) ||
    (c >= '0' as u8 && c <= '9' as u8)
}

// Check if all characters in a byte vector are valid
pub open spec fn all_valid_chars(s: Seq<u8>) -> bool {
    forall|i: int| 0 <= i < s.len() ==> #[trigger] is_valid_cookie_char(s[i])
}

// Serialize a single cookie to "key=value" format
fn serialize_cookie(key: &Vec<u8>, value: &Vec<u8>) -> (result: Vec<u8>)
    requires
        key.len() > 0,
        value.len() > 0,
        all_valid_chars(key@),
        all_valid_chars(value@),
    ensures
        result.len() == key.len() + 1 + value.len(),
{
    let mut result: Vec<u8> = Vec::new();

    // Copy key
    let mut i: usize = 0;
    while i < key.len()
        invariant
            i <= key.len(),
            result.len() == i,
    {
        result.push(key[i]);
        i = i + 1;
    }

    // Add '=' separator
    result.push('=' as u8);

    // Copy value
    let mut j: usize = 0;
    while j < value.len()
        invariant
            j <= value.len(),
            result.len() == key.len() + 1 + j,
    {
        result.push(value[j]);
        j = j + 1;
    }

    result
}

// Simple proof that serialized length equals key+1+value
```

```
proof fn lemma_serialize_length(key_len: nat, value_len: nat)
    ensures key_len + 1 + value_len > key_len,
{
    // Trivial arithmetic proof - the serialized form is longer than just the key
}

// Future extensions could include:
// - Parsing "key=value" strings back into Cookie structs
// - Handling multiple cookies separated by "; "
// - Validating cookie size limits (RFC6265 constraints)
// - Supporting cookie attributes like Path, Domain, Expires

} // verus!
```

Frama-C. Sampled repository: `edwtjo/nixitch`.

Frama-C example — sampled README

````
Repository: edwtjo/nixitch

# Project moved!

The main repo is at [NixOS](https://github.com/NixOS/nix-idea)
````

Frama-C example — sampled reference snippet 1

````
--- loop-invariant ---
A `loop invariant` is a property that holds before the loop and is preserved by every
     iteration. WP proves it on entry (establishment) and across one arbitrary iteration (
     preservation), then uses it together with the negated loop test to reason about the
     state after the loop. Strong enough invariants are the key to verifying loops.

```c
/*@
  requires n >= 0;
  assigns  \nothing;
  ensures  \result == n;
*/
int count_up(int n)
{
  int i = 0;
  /*@
    loop invariant 0 <= i <= n;
    loop assigns i;
    loop variant n - i;
  */
  while (i < n) {
    i++;
  }
  return i;
}
```
````

Frama-C example — sampled reference snippet 2

````
--- assertion ---
An `//@ assert` annotation states a predicate that must hold at a specific program point
     inside a function body. WP generates a goal to prove it from the facts available
     there, and the asserted fact then becomes available to subsequent reasoning.
     Assertions are useful as intermediate lemmas that guide the prover toward the
     postcondition.

```c
/*@
  requires 0 <= x <= 100;
  assigns  \nothing;
  ensures  \result == 2 * x;
*/
````
````

```
int double_it(int x)
{
  int y = x + x;
  //@ assert y == 2 * x;
  return y;
}
```

Frama-C example — Qwen2.5-Coder-32B response (verifies successfully)

```
/*
 * Nix Expression Path Validator
 *
 * Inspired by the Nix package manager's path expressions, this program
 * validates and manipulates simple path strings representing package derivations.
 * A valid path contains only alphanumeric characters, hyphens, and slashes,
 * and doesn't start/end with invalid characters.
 */

#include <stddef.h>

/*@ predicate is_valid_char(char c) =
      ('a' <= c <= 'z') || ('A' <= c <= 'Z') ||
      ('0' <= c <= '9') || c == '-' || c == '/';
*/

/*@ predicate valid_path{L}(char *p, integer len) =
      len >= 0 &&
      (\forall integer i; 0 <= i < len ==> is_valid_char(p[i])) &&
      (len == 0 || p[0] != '/') &&
      (len == 0 || p[len-1] != '/');
*/

/*@
  requires \valid_read(path + (0 .. len-1));
  requires len >= 0;
  assigns  \nothing;
  ensures  \result == 1 <==> valid_path(path, len);
  ensures  \result == 0 || \result == 1;
*/
int validate_nix_path(const char *path, int len)
{
  // Empty path is valid
  if (len == 0) {
    return 1;
  }

  // Check first character
  char first = path[0];
  if (first == '/') {
    return 0;
  }

  // Check last character
  char last = path[len - 1];
  if (last == '/') {
    return 0;
  }

  // Check all characters
  /*@
    loop invariant 0 <= i <= len;
    loop invariant \forall integer j; 0 <= j < i ==> is_valid_char(path[j]);
    loop assigns i;
    loop variant len - i;
  */
  for (int i = 0; i < len; i++) {
    char c = path[i];
    int valid = ((c >= 'a' && c <= 'z') ||
                 (c >= 'A' && c <= 'Z') ||
                 (c >= '0' && c <= '9') ||
                 c == '-' || c == '/');

    if (!valid) {
      return 0;
```

```
    }
  }

  return 1;
}

/*@
  requires \valid_read(path + (0 .. len-1));
  requires len >= 0;
  requires start >= 0 && start <= len;
  assigns  \nothing;
  ensures  \result >= start && \result <= len;
  ensures  \forall integer i; start <= i < \result ==> path[i] != '/';
  ensures  \result < len ==> path[\result] == '/';
  ensures  \result == len ==> (\forall integer i; start <= i < len ==> path[i] != '/');
*/
int find_next_slash(const char *path, int len, int start)
{
  /*@
    loop invariant start <= i <= len;
    loop invariant \forall integer j; start <= j < i ==> path[j] != '/';
    loop assigns i;
    loop variant len - i;
  */
  for (int i = start; i < len; i++) {
    if (path[i] == '/') {
      return i;
    }
  }
  return len;
}

/* Future extensions:
 * - Count path components (separated by slashes)
 * - Validate specific Nix attribute names (alphanumeric, hyphen, underscore)
 * - Parse and validate full Nix expression paths
 */
```