Improving autoformalization via cycle consistency and incremental type-checking using language-model probabilistic programs

Mauricio Barba da Costa[*1], **Fabian Zaiser**[*1], **Katherine M. Collins**[1], **Romir Patel**[1], **Timothy J. O'Donnell**[2,3,4], **Alexander K. Lew**[5], **Joshua B. Tenenbaum**[1], **Vikash K. Mansinghka**[1], **Cameron E. Freer**[1]

[1]MIT, [2]McGill University, [3]Mila – Québec AI Institute, [4]Canada CIFAR AI Chair, Mila [5]Yale University
{barba, fzaiser, katiemc, romir, jbt, vkm, freer}@mit.edu
timothy.odonnell@mcgill.ca, alexander.lew@yale.edu

Abstract

Autoformalization, the task of translating natural-language mathematics into a formal language such as Lean, has the potential to change the nature of mathematical discovery — providing mathematicians more certainty via verification during their proof-discovery process and even facilitating automated mathematical reasoning. In recent years, language models have made great strides towards autoformalization, but the task remains challenging both because of the rigidness of the target formal languages and the inherent uncertainty in informal texts. We propose a method for autoformalization via language-model probabilistic programming. Using tools for constrained generation from language models (LMs), we probabilistically steer LMs by incorporating two correctness signals: (i) incremental type-checking in Lean to rule out generations with no valid completion, and (ii) cycle consistency whereby a backtranslation of the formalized statement is constrained to be similar to the original informal statement. We demonstrate that both of these signals can improve autoformalization quality on the *miniF2F* and LeanEuclid *Book* datasets, while requiring fewer tokens and shorter runtime.

1 Introduction

The majority of human-written mathematics is expressed "informally" in natural language — using mathematical symbols and a variety of abstract notation and concepts, but nevertheless without formal semantics or the possibility of automatically checking the arguments. Formalization can turn informal mathematical statements and proofs into a concrete symbolic form that is verifiable, but writing proofs in formal mathematics is often far more tedious, slow, error-prone, and unnatural for human mathematicians [Bayer et al., 2024].

Autoformalization is the task of translating informal mathematical text into unambiguous statements in a machine-verifiable formal language used in a theorem prover such as Lean 4. Language models (LMs) are a natural candidate for this task due to their linguistic and code-generation abilities. However, such models suffer from several shortcomings: limited training data in Lean compared to common programming languages like Python, instability due to language and library changes, and the general problem of confabulation [Yang et al., 2025].

[*]Equal contribution.

The 5th Workshop on Mathematical Reasoning and Artificial Intelligence (MATH-AI), 2025.

To address these challenges, we propose leveraging techniques from language-model probabilistic programming [Lew et al., 2023, Loula et al., 2025] as a principled way of incorporating diverse correctness signals. In this paper, we illustrate how to use these tools to enforce type-checking and cycle consistency, and demonstrate that this boosts the quality of autoformalization. We find that across two datasets, both type-checking and cycle consistency can improve autoformalization quality.

More generally, our framework is flexible enough to incorporate a variety of semantic and syntactic constraints (enforcing mathematical correctness but also inferring plausibility and human intent), which can help in scaling autoformalization from single statements to longer texts, as well as for many other tasks in mathematical reasoning, including lemma synthesis, conjecturing, fixing incorrect statements, and premise selection.

We contribute: (1) **language model probabilistic programming for autoformalization**, a framework for using constrained generation via conditional inference to improve formalization quality. We use sequential Monte Carlo to sample from the posterior distribution of the LM distribution conditioned on the above signals of various kinds: binary (valid/invalid) vs. continuous, syntactic vs. semantic, hard (strict) vs. soft (lenient); (2) **incremental type-checking**: as one such signal, we run the Lean type-checker incrementally during LM generation, which allows us to detect and steer away from syntax- and type-errors as well as uses of deprecated/confabulated names for mathematical concepts earlier in the generation process; (3) **cycle consistency**: we adapt the classic idea of backtranslation from natural language translation to autoformalization using conditional probabilities of LM completions — specifically, asking an LM to informalize the candidate formalizations and score them according to how close they are to the original informal statement; (4) **empirical evaluation**: we demonstrate that our techniques can improve autoformalization performance, while using fewer tokens and less time.

2 Methods

2.1 Constrained Generation via Sequential Monte Carlo using GenLM

Constrained decoding has been used in the past to coerce language models to conform to a specific grammar. One of the first such iterations of this was done by Poesia et al. [2022], which constrains the output of each forward pass to tokens that are syntactically constrained and well-typed. However, locally constraining the next token does not generate sequences with the same probabilities as the language model conditioned on the desired constraints. Loula et al. [2025] and Lipkin et al. [2025] solve this problem by using sequential Monte Carlo methods to approximately sample from the correct conditional distribution, with Loula et al. [2025] also allowing for sampling from distributions weighted by continuous signals instead of strictly binary ones.

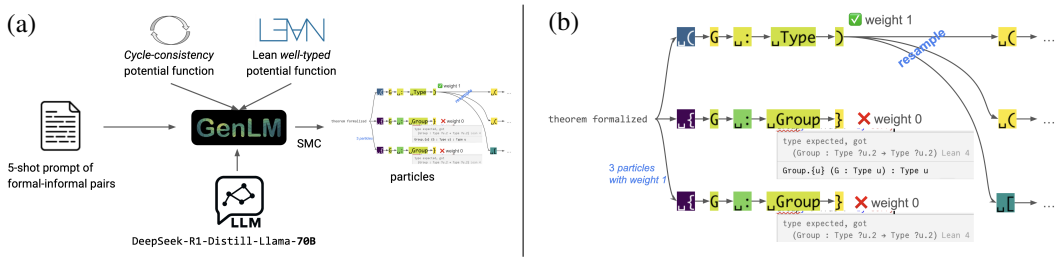


Figure 1: (a) GenLM performs sequential Monte Carlo inference to generate particles that are constrained from an LM given a prompt specifying the task and potentials for steering. (b) Sequential Monte Carlo guides LM sampling by generating, scoring, and resampling a number of weighted LM generations (*particles*). Their weights are adjusted according to the potentials (in this case, the Lean well-typed potential). Resampling (according to the weights) ensures particle diversity.

We use the *GenLM* [Loula et al., 2025] language-model probabilistic programming method. (For details, see Appendix A2.) Using it, we can program *potentials*, i.e., likelihood functions $\text{Token}^* \to [0, \infty)$, by which the prior is multiplied in order to "steer" the language model (Fig. 1a). Potentials can be *binary* (0 or 1), like checking syntactic validity, or *continuous*, like a numerical score assigned by another LM. We use GenLM to perform sequential Monte Carlo [Doucet et al., 2001] to sample approximately from the posterior distribution proportional to the prior LM distribution times the potentials (Fig. 1b).

2.2 Incremental Type-Checking

Running Lean's parser and type checker can detect various errors, such as syntax errors (e.g., LMs outputting Lean 3 code instead of Lean 4), semantic errors (e.g., nonexistent library concepts confabulated by the LM), and type errors (e.g., invalid composition of concepts by the LM). It is desirable to catch such errors as early as possible; as such, we run the Lean type checker incrementally at "safe points" during LM generation, such as just after a completed hypothesis or conclusion. At these points, the Lean theorem statement is incomplete, so Lean's parser would always yield an error. To prevent this, we complete the generated theorem with a "dummy" conclusion and proof (see Fig. 2). This Lean well-typed potential is active only within ```lean4 code blocks in the LM's output and rejects generations whose "dummy" completion fails to type-check.

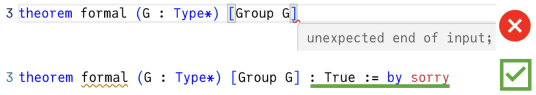


Figure 2: Incremental type-checking: during LM generation, we run the Lean type checker after every completed hypothesis (in this example, `[Group G]`) and conclusion. Running the type checker directly would lead to a syntax error, so we add a dummy conclusion (`True`) and proof (`by sorry`), which is accepted by Lean.

2.3 Cycle Consistency

The *cycle consistency* heuristic, whereby translations are judged to be more accurate when they have backtranslations similar to the original, has been used in image processing [Zhu et al., 2017], natural language [Lample et al., 2018], and autoformalization [Wu et al., 2022, Gao et al., 2025, Li et al., 2024]. In Fig. 3, we adapt this method to our setting using an LM to auto**in**formalize a candidate formalization. As part of our SMC pipeline, we generate multiple particles for each formalization attempt in parallel. We then score these particles by the log probability of the LM generating the original statement given the candidate formalization. Insofar as our method samples multiple particles in parallel and scores the particles at the end, it is similar to Li et al. [2024], but we score particles using the log probability instead of cosine similarity and we don't use an automatic theorem-prover to assess correctness. Our method uses only the base language model and, unlike other autoformalization methods, does not require any training.

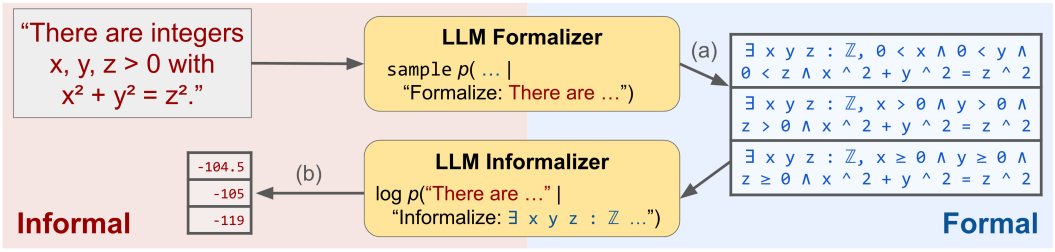


Figure 3: Cycle consistency prefers formalizations whose informalizations are close to the original. Step (a) samples from the distribution of an LM (`DeepSeek-R1-Distill-Llama-70B`) which is prompted to formalize the input (optionally using controlled generation). Step (b) scores the formalizations according to the log probability that the LM generates the original input text when it is prompted to informalize the given formalization. Inaccurate formalizations (like the third, which uses "≥ 0" instead of "> 0") rank lower.

2.4 Autoformalization using GenLM

We combine all these ingredients to improve the autoformalization accuracy of the base LM. We prompt `DeepSeek-R1-Distill-Llama-70B` to translate a given natural-language mathematics statement to a Lean 4 statement after a 5-shot prompt of formal-informal example pairs. We use GenLM

Table 1: **The Lean well-typed and cycle-consistency potentials each separately improve autoformalization quality, and their combination performs best — while using fewer tokens and achieving shorter runtime than the baseline.** Both the baseline and our method use `DeepSeek-R1-Distill-Llama-70B`. The baseline is a single run, with unrestricted CoT. Our method uses GenLM to run SMC with 5 particles, with CoT suppressed. Our full method appears in the "both potentials" row, with three ablations shown above.

	miniF2F			LeanEuclid *Book*		
Method	Score (%)	Tokens	Time (s)	Score (%)	Tokens	Time (s)
Baseline: single run with CoT	16.0	1715	124.4	14.0	1040	92.1
Ours: GenLM running SMC with						
no potential	25.6	252	7.5	7.9	552	15.9
Lean well-typed potential	34.1	952	49.7	14.2	547	16.6
cycle-consistency potential	32.7	252	7.5	4.8	552	15.9
both potentials	**38.0**	952	49.7	**16.8**	547	16.6

to run sequential Monte Carlo on this prior distribution with the Lean well-typed potential (including incremental type-checking) and the cycle-consistency potential. (For details on potentials, see Appendix A3.)

3 Empirical Evaluation

We benchmark our autoformalization performance on two challenging datasets (see Table 1). The *miniF2F* dataset consists of high school and college-level mathematics problems. The LeanEuclid *Book* dataset from Murphy et al. [2024] contains Euclidean geometry problems stated in a constrained domain-specific language (DSL), in which base LMs have not been substantially pretrained. We use the 43 problems in the *Book* dataset and subsample 50 problems from the *miniF2F* dataset.

As a baseline, we run `DeepSeek-R1-Distill-Llama-70B` once. Our method uses GenLM to run SMC with 5 particles using the same LM with chain-of-thought (CoT) suppressed, using both the Lean well-typed and cycle-consistency potentials, which we label "both potentials" in Table 1. We also consider ablations of our full model (removing each potential). We label these ablations "no potential", "Lean well-typed potential", and "cycle-consistency potential".

We evaluate the empirical distribution of the correctness of formalizations obtained by the 5 particles. For LeanEuclid *Book*, automated equivalence checks are possible, and so to judge correctness we use the symbolic checker E3, which verifies whether a candidate formalization is logically equivalent to the ground truth in the constrained geometry DSL, as in Murphy et al. [2024]. For *miniF2F*, no symbolic tool for correctness is available, and so we rely on hand annotation to assess the semantic match (0% or 100%) between the candidate formalization and the informal statement. The score for a given formalization task is the weighted average of each particle's correctness score. These scores are then averaged across all 50 problems for the dataset. (For further details, see Appendix A1.)

Our method improves autoformalization quality on both *miniF2F* and LeanEuclid *Book* datasets, while requiring substantially fewer tokens and less wall-clock time. Both the Lean well-typed potential and the cycle-consistency potential contribute to this, as shown by our ablation study.

4 Discussion

Our results demonstrate the promise of language-model probabilistic programming for autoformalization. Both Lean well-typed and cycle-consistency potentials improve the autoformalization performance, with the combination of both achieving the best performance — while using fewer tokens and achieving shorter runtime. This could allow us to lift smaller models towards the quality of larger baselines, which might enable compute-efficient locally-runnable AI thought partners [Collins et al., 2024, Frieder et al., 2024] for mathematicians.

More generally, sequential Monte Carlo can be used to steer an LM using a wide range of customizable potentials, encoding priors about the likely intent of mathematicians in the informal text. By incorporating a richer array of such priors (including retrieval from Mathlib, common patterns in mathematical statements, plausibility in light of earlier text, etc.), a wide range of human mathematical behavior can be captured as inference over the same base LMs. Autoformalization systems that are attentive to these subtleties in informal text could free mathematicians to continue to explore mathematics informally while keeping their work verifiable.

Our research also has implications for programming language design. In particular, this method of constraining an LM via incremental type-checking could be generally useful in strongly-typed functional languages, and cycle consistency could be useful in cross-compilation.

Acknowledgments

We thank Benjamin LeBrun, Benjamin Lipkin, Ced Zhang, and the GenLM Consortium for valuable conversations that informed this work. KMC acknowledges support from the Cambridge Trust and King's College Cambridge.

References

Jonas Bayer, Christoph Benzmüller, Kevin Buzzard, Marco David, Leslie Lamport, Yuri Matiyasevich, Lawrence Paulson, Dierk Schleicher, Benedikt Stock, and Efim Zelmanov. Mathematical proof between generations. *Notices of the American Mathematical Society*, 71(1):79–92, 2024. URL https://doi.org/10.1090/noti2860.

Katherine M. Collins, Ilia Sucholutsky, Umang Bhatt, Kartik Chandra, Lionel Wong, Mina Lee, Cedegao E. Zhang, Tan Zhi-Xuan, Mark Ho, Vikash Mansinghka, Adrian Weller, Joshua B. Tenenbaum, and Thomas L. Griffiths. Building machines that learn and think with people. *Nature Human Behaviour*, 8(10):1851–1863, 2024. URL https://doi.org/10.1038/s41562-024-01991-9.

Arnaud Doucet, Nando de Freitas, and Neil J. Gordon. An introduction to sequential Monte Carlo methods. In Arnaud Doucet, Nando de Freitas, and Neil J. Gordon, editors, *Sequential Monte Carlo Methods in Practice*, Statistics for Engineering and Information Science, pages 3–14. Springer, 2001. URL https://doi.org/10.1007/978-1-4757-3437-9_1.

Simon Frieder, Jonas Bayer, Katherine M. Collins, Julius Berner, Jacob Loader, András Juhász, Fabian Ruehle, Sean Welleck, Gabriel Poesia, Ryan-Rhys Griffiths, Adrian Weller, Anirudh Goyal, Thomas Lukasiewicz, and Timothy Gowers. Data for mathematical copilots: Better ways of presenting proofs for machine learning, 2024. URL https://arxiv.org/abs/2412.15184.

Guoxiong Gao, Yutong Wang, Jiedong Jiang, Qi Gao, Zihan Qin, Tianyi Xu, and Bin Dong. Herald: A natural language annotated Lean 4 dataset, 2025. URL https://arxiv.org/abs/2410.10878.

GenLM Consortium. GenLM-Control Library, 2025. URL https://genlm.org/genlm-control.

Guillaume Lample, Alexis Conneau, Ludovic Denoyer, and Marc'Aurelio Ranzato. Unsupervised machine translation using monolingual corpora only. In *International Conference on Learning Representations (ICLR 2018)*, 2018. URL https://openreview.net/forum?id=rkYTTf-AZ.

Alexander K. Lew, Tan Zhi-Xuan, Gabriel Grand, and Vikash K. Mansinghka. Sequential Monte Carlo steering of large language models using probabilistic programs, 2023. URL https://arxiv.org/abs/2306.03081.

Zenan Li, Yifan Wu, Zhaoyu Li, Xinming Wei, Xian Zhang, Fan Yang, and Xiaoxing Ma. Autoformalize mathematical statements by symbolic equivalence and semantic consistency, 2024. URL https://arxiv.org/abs/2410.20936.

Benjamin Lipkin, Benjamin LeBrun, Jacob Hoover Vigly, João Loula, David R. MacIver, Li Du, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Timothy J. O'Donnell, Alexander K. Lew, and Tim Vieira. Fast controlled generation from language models with adaptive weighted rejection sampling. In *Proceedings of the 2nd Conference on Language Modeling (COLM 2025)*, Montréal, Canada, 2025. URL https://openreview.net/forum?id=3BmPSFAdq3.

João Loula, Benjamin LeBrun, Li Du, Ben Lipkin, Clemente Pasti, Gabriel Grand, Tianyu Liu, Yahya Emara, Marjorie Freedman, Jason Eisner, Ryan Cotterell, Vikash Mansinghka, Alexander K. Lew, Tim Vieira, and Timothy J. O'Donnell. Syntactic and semantic control of large language models via sequential Monte Carlo. In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR 2025)*, Singapore, April 2025. URL https://openreview.net/forum?id=xoXn62FzD0.

Wenjie Ma, Jingxuan He, Charlie Snell, Tyler Griggs, Sewon Min, and Matei Zaharia. Reasoning models can be effective without thinking, 2025. URL https://arxiv.org/abs/2504.09858.

Logan Murphy, Kaiyu Yang, Jialiang Sun, Zhaoyu Li, Anima Anandkumar, and Xujie Si. Autoformalizing Euclidean geometry. In *Proceedings of the 41st International Conference on Machine Learning (ICML 2024)*, volume 235 of *Proceedings of Machine Learning Research*, pages 36847–36893. PMLR, 2024. URL https://proceedings.mlr.press/v235/murphy24a.html.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. Synchromesh: Reliable code generation from pre-trained language models. In *International Conference on Learning Representations (ICLR 2022)*, 2022. URL https://openreview.net/forum?id=KmtVD97J43e.

Yuhuai Wu, Albert Q. Jiang, Wenda Li, Markus N. Rabe, Charles Staats, Mateja Jamnik, and Christian Szegedy. Autoformalization with large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NeurIPS '22, 2022. URL https://proceedings.neurips.cc/paper_files/paper/2022/file/d0c6bc641a56bebee9d985b937307367-Paper-Conference.pdf.

Kaiyu Yang, Gabriel Poesia, Jingxuan He, Wenda Li, Kristin E. Lauter, Swarat Chaudhuri, and Dawn Song. Position: Formal mathematical reasoning — a new frontier in AI. In *Forty-second International Conference on Machine Learning (ICML 2025) Position Paper Track*, 2025. URL https://openreview.net/forum?id=HuvAM5x2xG.

Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. minif2f: A cross-system benchmark for formal olympiad-level mathematics. In *International Conference on Learning Representations (ICLR 2022)*, 2022. URL https://openreview.net/forum?id=9ZPegFuFTFv.

Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017. URL https://openaccess.thecvf.com/content_iccv_2017/html/Zhu_Unpaired_Image-To-Image_Translation_ICCV_2017_paper.html.

Appendix

A1 Experiment Details **8**

A1.1 Suppressing Chain-of-Thought Tokens 8

A1.2 Prompt 8

A1.3 Datasets 9

A1.4 Computational Resources 9

A1.5 Transpilation for the LeanEuclid *System E* DSL 9

A1.6 Token Usage 9

A1.7 Examples 10

A2 GenLM Background **11**

A3 Potentials **11**

A3.1 Lean Well-Typed Potential 12

A3.2 Cycle-Consistency Potential 12

A1 Experiment Details

A1.1 Suppressing Chain-of-Thought Tokens

Our experiments use `DeepSeek-R1-Distill-Llama-70B`, a distillation of a reasoning model. Although this is among the more capable LMs of its size for our autoformalization task, the expensive thinking traces (often thousands of tokens) that it outputs by default often do not materially improve the formalizations. We use the method described by Ma et al. [2025] to force the LM to output Lean code without a thinking trace. In particular, we append `<think>Okay, I'm done thinking.</think>\n```lean4\n` to our LM prompts, which typically leads the LM to output merely a completion of the empty `lean4` code block.

A1.2 Prompt

Our prompt has the following general format, where the text through "..." consists of several example natural-language/formal pairs taken from the respective *miniF2F* or LeanEuclid *Book* dataset, followed by the candidate informal text we wish the LM to formalize. In the case of *Book*, we also include a basic description from the LeanEuclid paper (because it is a DSL that the LM has likely not seen before).

```
|START_USER_CHAT|
Natural language version: Let X be a topological space; let A be a subset of
X. Suppose that for each x ∈ A there is an open set U containing x such that
U ⊂ A. Then A is open in X.

Translate the natural language version to a Lean 4 version:
import Mathlib
theorem subset_of_open_subset_is_open (X : Type*) [TopologicalSpace X] (A :
Set X) (hA : ∀x ∈ A, ∃U : Set X, IsOpen U ∧ x ∈ U ∧ U ⊆ A): IsOpen A := by
sorry
```

...

```
Natural language version: If x is an element of G such that x^n ≠ 1 for all
n, prove that the elements x^n are all distinct.

Translate the natural language version to a Lean 4 version:
|END_USER_CHAT|
|START_ASSISTANT_CHAT|
<think>
Okay, I'm done thinking.
</think>
```lean4
```

A1.3 Datasets

We benchmark our results on two datasets: 50 problems from *miniF2F* [Zheng et al., 2022] and 43 problems from LeanEuclid *Book* [Murphy et al., 2024]. The *miniF2F* dataset has informal and corresponding formalized mathematics problems from high-school mathematics and olympiads and undergraduate mathematics. We use the *Book* benchmark from LeanEuclid, which contains formalizations of statements from Euclid's Elements in a Lean DSL for synthetic geometry.

A1.4 Computational Resources

Running our experiments on the *miniF2F* and LeanEuclid *Book* datasets (approximately 50 problems each) took roughly 6 hours with 4 A100 GPUs (80 GB).

A1.5 Transpilation for the LeanEuclid *System E* DSL

Theorem statements in LeanEuclid's *System E* DSL must be of the form $\forall ... \to \exists ....$ All variables and hypotheses must be defined in the universal quantifier, which is more constrained than the way that Lean is usually structured:
`theorem example_thm (h1 : ...) ... : ∃ ... := by sorry`
Since many language models are trained on this type of Lean statement formatting, we convert the latter format to the former.

A1.6 Token Usage

We find that the tokens-per-second rate with or without our potentials are similar for the LeanEuclid *Book* benchmark. Running the *miniF2F* benchmarks with the Lean well-typed potential takes considerably longer than the other benchmarks because constraining the LM sometimes causes it to enter loops that lead the LM to output nonsensical tokens until it reaches the max token limit of 500. Strangely, Table 2 shows that the token per second rate of running the *miniF2F* benchmark with the Lean well-typed potential also decreases. We hypothesize that this is due to KV cache overhead on our hardware. Our observations regarding token usage are generally aligned with Ma et al. [2025], which finds that when controlling for the number of tokens, parallel sampling schemes that suppress thinking and aggregate multiple outputs can outperform language models with reasoning.
```

Table 2: **Incremental type-checking with the Lean well-typed potential adds little overhead on *miniF2F* and LeanEuclid *Book* benchmarks.** "Base" shows the number of tokens generated per second and "parallelism-adjusted" indicates that we divide the values by 5 to account for the tokens that are generated in parallel by GenLM which do not contribute to making the sequence longer.

	miniF2F		LeanEuclid *Book*	
Method	Base	Parallelism-adjusted	Base	Parallelism-adjusted
Baseline: single run with CoT	13.8	—	11.3	—
No potential	33.6	6.7	34.7	6.94
Lean well-typed potential	19.2	3.8	33.0	6.6

A1.7 Examples

A common failure mode of language models when producing Lean code is that they confabulate the names of lemmas and they call functions with improper types. Both of these errors are picked up by our incremental type-checking system. Below are examples of some of the base language model producing ill-typed code that is mitigated with GenLM.

```
Formalization: ∀ (a : Point) (b : Point) (c : Point) (d : Point) (e : Point)
(f : Point) (AB : Line) (BC : Line) (AC : Line) (DE : Line) (DF : Line)
(EF : Line), ((formTriangle a b c AB BC AC) ∧ (formTriangle d e f DE EF DF)
∧ (|(a-b)| = |(d-e)|) ∧ (|(a-c)| = |(d-f)|) ∧ (∠ a:b:c = ∠ d:e:f)) → ((
(|(b-c)| = |(e-f)|) ∧ (Triangle.congruent △ a:b:c △ d:e:f) ∧ (∠ b:c:a = ∠
e:f:d) ∧ (∠ a:c:b = ∠ d:f:e)))

Error: function expected at c term has type Point
```

```
Formalization: ∀ (a : Point) (b : Point) (c : Point) (d : Point) (AB : Line)
(BC : Line) (AC : Line), ((formTriangle a b c AB BC AC) ∧ (Point.between b c
d)) → (∠ a:c:d > ∠ b:a:c ∧ ∠ a:c:d > ∠ a:b:c)

Error: unknown constant 'Point.between'
```

The cycle-consistency potential improves the performance of our model by upweighting generations that resemble the informal text more closely. The example below shows a natural language statement in the *miniF2F* dataset followed by 4 formalizations that were generated in parallel using our method. Initially, all formalizations are assigned roughly equal probability. With cycle consistency, however, the probability of a correct formalization being sampled increases from 50% to 100%.

```
Given f(x) = cx^3 − 9x + 3 and f(2) = 9, find the value of c. Show that it is 3.
```

```
theorem formalized_thm (x : ℝ) (c : ℝ) (h₁ : ∀ x, x = 2 → cx^3 − 9x + 3 = 9)
: c = 3 := by

Status: Incorrect
Original probability: 0.25
Probability with cycle consistency: 0.00
```

```
theorem formalized_thm (f : ℝ → ℝ) (c : ℝ) (h : ∀x, f x = cx^3 − 9x + 3) (hf
: f 2 = 9) : c = 3 := by

Status: Correct
Original probability: 0.25
Probability with cycle consistency: 0.26
```

```
theorem formalized_thm : (3 : ℝ) = 3 := by

Status: Incorrect
Original probability: 0.25
Probability with cycle consistency: 0.00
```

```
theorem formalized_thm (c : ℝ) (f : ℝ → ℝ := fun x ↦ cx^3 − 9x + 3) (h : f 2
= 9) : c = 3 := by

Status: Correct
Original probability: 0.25
Probability with cycle consistency: 0.73
```

A2 GenLM Background

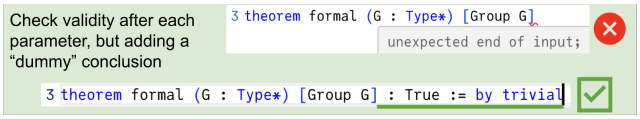
Interface to GenLM The GenLM library [GenLM Consortium, 2025] samples from distributions over language model completions weighted by certain functions known as *potentials*. The distribution given by a language model's distribution over completions p and a potential Φ is given by

$$g(\mathbf{x}) = \frac{1}{Z} p(\mathbf{x}) \Phi(\mathbf{x}).$$

The overall potential Φ we use here is a product of the following components, which we describe in more detail in Appendix A3 below.

1. Well-typedness, Φ_{WT}: The potential is 1 when the output is well-typed and is 0 otherwise.
2. Cycle consistency, Φ_{CC}: The potential is given by the likelihood of the *informal statement* given by a small language model where the prompt is to informalize the formal statement.

SMC Steering A key advantage of sequential Monte Carlo is its ability to incrementally update posterior estimates in light of newly observed text. We incrementally type-check Lean code by filling in intermediate hypotheses with a dummy conclusion and proof.



Adaptive Weighted Rejection Sampling The specific SMC sampling procedure we use via GenLM is known as Adaptive Weighted Rejection Sampling (AWRS) [Lipkin et al., 2025].

A3 Potentials

GenLM potentials must define both `prefix` and `complete` methods, and can be either *binary* ($\{0, 1\}$-valued) or *continuous* ($[0, 1]$-valued).

A3.1 Lean Well-Typed Potential

The Lean well-typed potential Φ_{WT} is binary ($\{0, 1\}$-valued), and aims to reject a wide variety of strings that cannot possibly be extended in a way that makes them well-typed in Lean, without rejecting any that admit at least one well-typed completion.

On a complete LM generation ("complete"), the Lean well-typed potential runs the Lean parser and type checker. If there are no errors, the potential accepts (returns 1), otherwise rejects (returns 0).

On an incomplete LM generation ("prefix"), the Lean well-typed potential first rejects comments and line breaks (as LMs sometimes use them to "put off" the salient part of the formalization). Next, it checks whether the current prefix is at a likely boundary, i.e., whether it is just after a completed hypothesis (ending with `)`, `]`, or `}`) or conclusion (ending with `:=`), which can be checked for well-typedness. It parses the generated Lean code, replacing proofs with `sorry` (as we only care about statement formalization) and patches incomplete syntax, by filling in missing syntactic elements with "dummy" values (`True` for the conclusion and `sorry` for the proof). Finally, the Lean parser and type checker is run, just like for `complete`.

```
def prefix(context):

  if empty(context):
    return 1

  generated_text = tokens_to_str(context)
  if comments or newlines found:
    return 0

  if not at likely boundary: # not ending with ) ] } := or ;
    return 1

  text = lean_prefix + generated_text
  commands = parse_lean(text)
  remove proofs from theorems
  patched = patch_syntax(commands)

  return 1 if is_well_typed_lean(patched) else 0

def complete(context):

  if empty(context):
    return 1

  text = lean_prefix + tokens_to_str(context)

  return 1 if is_well_typed_lean(text) else 0
```

Listing 1: Pseudocode for Lean well-typed potential

A3.2 Cycle-Consistency Potential

For complete generations, the potential Φ_{CC} informalizes the output by prompting a different LM (in our case, `Llama-3.2-3B` for speed and smaller memory footprint) to translate the formalization candidate back to natural language. (This "backwards" task is generally easier for LMs than the forward direction of formalization.) The potential computes the probability that the LM produces the original informal statement as a response to that prompt. This probability is a continuous ($[0, 1]$-valued) signal about the quality of the formalization since a faithful formalization should contain all the information in the original.

```
def prefix(context):
    # No prefix constraints/scoring in this potential
    return 1

def complete(context):
    if empty(context):
        return 1
    formalized_output = tokens_to_str(context)
    prompt = informalization_prompt(formalized_output) # essentially:
        "Informalize: {formalized_output}"
    lm.set_prompt_from_str(prompt)
    # Return the probability of producing the original informal
        statement
    return lm.probability(informal_statement)
```

Listing 2: Pseudocode for cycle-consistency potential