Feedback & Synthesis in LLM-Assisted Termination Proofs

Zeke Medley
Khoury College of Computer Sciences, Northeastern University, Boston, MA, USA

Panagiotis Manolios
Khoury College of Computer Sciences, Northeastern University, Boston, MA, USA

Abstract

Termination – proving there are no inputs on which a function runs forever – is one of the most fundamental problems in software verification, and competitions comparing termination analysis tools have run for over twenty years. We integrate two state-of-the-art, open-weight LLMs with a theorem prover's built-in automation to generate termination proofs, solving 39% more problems than the best LLM alone and more than tripling the number solved by the built-in analysis on our benchmark. This is the first, to our knowledge, integration of an LLM with a termination-analysis algorithm, and is generalizable to any theorem prover based on a functional language. Our design is informed by nine ablations considering what theorem-prover feedback helps the LLM and four experiments on how the model decomposes problems.

2012 ACM Subject Classification Software and its engineering → Formal software verification; Theory of computation → Program reasoning; Computing methodologies → Artificial intelligence

Keywords and phrases termination analysis, theorem proving, large language models, ACL2

Digital Object Identifier 10.4230/LIPIcs.ITP.2026.22

Supplementary Material *Software (Source Code)*: `https://github.com/0xekez/fslatp` [31] archived at `swh:1:dir:26cbefe528532dda47d04b7198cca383281e5a72`

Declaration about GenAI/LLM use The code was handwritten with help from an LLM.

1 Introduction

Recently, LLMs have had remarkable success writing computer-checkable formal proofs [1, 10, 16, 25, 37, 39, 45, 46]. Two models [1, 10], for example, achieved gold-medal-level performance on the 2025 International Mathematical Olympiad with solutions formalized in Lean [14]. AxiomProver [4] and Aristotle [1] scored 9/12 and 10/12 respectively on the 2025 Putnam Competition, also formalizing solutions in Lean. In response, a promising new line of work [7, 13, 38, 41, 51] considers if the mathematical capabilities of LLMs might translate to software verification.

One of the most fundamental problems in software verification is termination. The Lean [14], Rocq [40], and ACL2 [22] theorem provers require all admitted logical functions to be proven terminating, and terminating functions give rise to induction schemes widely used to prove properties of recursive functions. As such, a considerable amount of research [11, 17, 28] considers how to automatically prove functions terminate with the Annual International Termination Competition [30] held since 2004 to evaluate these tools.

This paper investigates how `gpt-oss-120b` [35] and `Qwen3-32B` [50], two state-of-the-art open-weight models, prove functions terminate using the ACL2s [15] theorem prover. We ask three questions: **Q1** How good are the models at proving functions terminate? **Q2** What theorem-prover feedback is useful? **Q3** Can the models be integrated synergistically with the theorem-prover's automated termination analysis? Focusing on two models lets us look closely at these questions, the experiments here requiring 786 hours on an H200 GPU. Code for reproducing and extending our results is publicly available in § 11.

The feedback question is interesting because, despite being state-of-the-art at the time of their release, neither DeepSeek-Prover v1, v1.5, nor v2 [39, 48, 49] provides theorem-prover error messages to the LLM during proof generation, nor does Godel-Prover [25], and others like Kimina-Prover [46] employ an explicit training step to encourage the model to use them. It is only more recently that models like SeedProver [10] and Aristotle [1] began integrating theorem-prover feedback. The success of these earlier models despite not using feedback raises questions about how and to what extent it is useful.

As our evaluation dataset, we curate a base corpus of interesting termination problems from a university course on program verification [27] and past termination competitions [30]. Then, we develop three ways to synthetically generate terminating functions from our base corpus and evaluate the model on these. Previous work [32] has shown that adding inconsequential details to problems from a math benchmark [33] can degrade performance, suggesting models use memorized patterns rather than genuine reasoning. Our synthetic functions being novel help guard against memorization.

At a glance, our findings are as follows.

Q1 The models can solve difficult termination problems, considerably outperforming ACL2s' built-in analysis [28]. However models struggle with problem decomposition, often failing to prove termination of trivial combinations of functions whose termination it can prove individually.

Q2 The models use all levels of theorem-prover feedback from syntax errors to proof failure diagnostics. However, including a history of previous attempts and errors does not always improve performance. For example, `gpt-oss-120b` does better when shown only its most recent failed attempt than when shown a complete history of attempts and their corresponding errors. Moreover, the best-performing configuration for both models does not include any information about the history of previous attempts. More theorem-prover feedback is not always better.

Q3 While the set of problems solved by the model is considerably larger than ACL2s' built-in analysis, it is not a superset. When the built-in analysis analyzes a function, it decomposes it into its individual loops; we suspected this might help the model with decomposition, therefore, our best theorem-prover LLM integration provides LLM-generated proofs these inner loops terminate to the built-in analysis as a hint. This considerably outperforms the model and built-in analysis alone, demonstrating that existing proof automation techniques can work synergistically with an LLM.

The paper is organized as follows. Section 2 reviews related work in termination analysis and LLM-assisted theorem proving. Section 3 describes the integration of the models with the ACL2s theorem prover and the design of the feedback mechanism. Section 4 details the model configurations and experimental environment. Section 5 introduces the base problem corpus, and Section 6 details how synthetic problems are generated from it. Section 7 presents the results of nine ablation experiments on the feedback mechanism. Section 8 describes an integration of ACL2s' built-in termination analysis with an LLM, solving 39% more problems than the best LLM alone. Section 9 experimentally investigates the model's underperformance decomposing termination problems. Section 10 concludes and gives directions for future research.

In summary, our contributions are as follows.

1. A method for integrating an LLM with automated termination analysis which solves 39% more problems than the LLM alone on our benchmark.
2. The first, to our knowledge, evaluation of an LLM on termination proofs, and a method for synthetically generating challenging termination problems.

3. Through nine ablations, an evaluation of the impact of theorem-prover feedback on LLM performance. While feedback on proof failures improves performance, providing a history of past attempts often degrades it. Our best configuration uses error feedback without history.
4. Experimental results that the model fails to reliably decompose termination problems, even when the problem structure is explicitly revealed. This justifies our architecture where the theorem prover handles decomposition and the LLM solves the resulting sub-problems.

2 Related Work

A common design choice across theorem prover lineages is the requirement that all functions terminate. This decision buys nice meta-theory, a powerful relationship between recursive functions and induction, and makes proving function termination one of the most fundamental problems in theorem proving.

In the Boyer–Moore theorem prover tradition (NQTHM [6] → ACL2 [22] → ACL2s [15]) a recursive function f is only admitted once a well-founded relation between its successive recursive calls has been proven to exist [22, 27]. A relation over a set X is well-founded if it contains no infinite decreasing sequences. In the termination context where X denotes recursive calls to f, the existence of a well-founded relation means there is no infinite sequence of recursive calls to f, and in turn f terminates on all inputs.

For example, consider the function

```
(definec f (x y :int) :tl
  (if (< x 0)
      ()
    (f (+ x y) (- y 1))))
```

To prove f terminates, we can define a measure $m(x, y) = (y + 1)^2 + 2x$ and note that on a recursive call to f, m decreases.

$$m(x, y) - m(x + y, y - 1) = 1$$

When a recursive call to f occurs, $x \geq 0$, so m maps successive recursive calls to f to a decreasing sequence over the natural numbers, there are no infinite, decreasing sequences over the natural numbers, so m gives rise to a well-founded relation between successive recursive calls to f, and in turn f is proven to terminate.

The theorem-proving power of terminating functions comes from the induction schemes they give rise to. For example, to prove φ using the induction scheme of f our proof obligations become:

1. $(\neg\texttt{(intp } x\texttt{)} \vee \neg\texttt{(intp } y\texttt{)}) \rightarrow \varphi$
2. $(\texttt{(intp } x\texttt{)} \wedge \texttt{(intp } y\texttt{)} \wedge x < 0) \rightarrow \varphi$
3. $(\texttt{(intp } x\texttt{)} \wedge \texttt{(intp } y\texttt{)} \wedge x \geq 0 \wedge \varphi[x \coloneqq x + y,\ y \coloneqq y - 1]) \rightarrow \varphi$

Where `(intp` x`)` is true if x is an integer. For more information about how terminating functions give rise to induction schemes, see Part V of *Reasoning About Programs* [27]. These induction schemes are one of the most powerful tools available in theorem provers, making proving function termination one of the most important skills for users, and motivating this paper's study of how well LLMs prove function termination.

While we focus on the ACL2s theorem prover, provers in the CIC/MLTT lineage (Rocq, Agda, Lean) likewise require all logical definitions to be terminating to maintain consistency. Although this requirement is justified by strong normalization at the level of the core calculus, termination is enforced operationally through structural and well-founded recursion checks. Rocq utilizes structural recursion checks and well-founded recursion [12, 36]. Agda supports structural termination through sized types [2], and Lean 4 provides `termination_by` and `decreasing_by` for proving termination [24]. The underlying logics differ, but the challenge of identifying a decreasing measure is shared across all these systems.

2.1 Termination Cores

ACL2s' built-in termination analysis constructs an overapproximation of program behaviors called the calling context graph (CCG) [28]. If it cannot prove termination automatically, it analyzes the CCG to find one simple cycle which it cannot prove terminates, called a termination core [29]. The termination core is surfaced to the user, who can provide hints to the analysis to prove the core terminates. For example, consider the function `f` below.

```
(definec f (x y :nat) :nat
  (if (and (< x y) (> x 0))
      (f (* 3 x) (* 2 y))
    (if (> x y)
        (f x (+ y 1))
      y)))
```

`f` consists of two recursive calls: f_1, which runs when x is less than y and x is greater than 0, multiplies x by 3 and y by 2; and f_2, which runs when x is greater than y, adding 1 to y. f's calling context graph has vertices f_1 and f_2, and an edge between two vertices if ACL2s cannot prove[1] a call of one cannot result in a call of the other. For f, this means there is no edge from f_2 to f_1, as when y is less than x adding 1 to y will never make it greater than x, yielding the CCG below.

Next, ACL2s analyzes the CCG using the Size-Change Termination (SCT) principle. It constructs size-change graphs for the edges and verifies that every infinite path through the CCG contains a thread of measures that decrease infinitely often and never increase, typically by analyzing idempotent matrices or performing a global graph analysis. If such a decreasing thread is found for all infinite paths, the function's termination is proven. Simple cycles that ACL2s cannot automatically prove the termination of are called *termination cores*. When ACL2s tries to admit f, it cannot prove the termination of the cycle $f_1 \to f_1$, so the termination core below is surfaced to the user.

```
(definec f_1 (x y :nat) :all
  (if (and (< x y) (< 0 x))
      (f_1 (* 3 x) (* 2 y))
    (list x y)))
```

Optimistically, if the function is actually terminating, the user then provides a measure for the termination core, and f is proven terminating. Later in this paper (§ 9), we will experimentally notice `gpt-oss-120b` struggles to decompose problems: often failing to prove

[1] This is the sense in which the CCG is an overapproximation of the program's behavior.

the termination of a combination of functions, despite successfully proving the termination of each individually. We will then integrate the model with ACL2s' ability to decompose problems into termination cores to considerable success.

2.2 LLMs & Formal Theorem Proving

Computer-checked formal proofs have an extremely high bar for correctness, making them resistant to hallucinations [20] and other inaccuracies that typically plague LLMs. This has led to considerable research on training LLMs to generate formal proofs, as they present a well-defined objective and, at least in humans, might transfer to general reasoning abilities [23,47]. Results on training models to write formal proofs are promising, for example, on MiniF2F [52] (a collection of Olympiad-level, formalized math problems), DeepSeek-Prover-V1.5 [49] improves from 29.7% to 51.6% accuracy (Pass@128[2]) on the test split after fine-tuning and reinforcement learning. Many other success stories, including ones mentioned in the introduction, exist in the literature [1, 10, 16, 25, 37, 39, 45, 46].

Building off the success of LLMs in mathematical theorem proving, several works [7, 13, 38, 41, 51] have considered whether LLMs can prove theorems about computer programs. These results are also promising, and good LLM theorem-prover integrations have potential to decrease the amount of manual effort required to verify software, a significant barrier to wider adoption of formal verification [18].

For termination in particular, recent work [42, 43] has used LLMs to predict if a program is terminating or not. With test-time-scaling, the authors found GPT-5 [34] and Claude Sonnet-4.5 [3] would rank just behind the top-ranked tool in the Termination category of the International Competition on Software Verification [5] 2025. Though, the authors did not actually enter the tools into the competition; timeouts and hardware limitations would likely be an issue if they did. Furthermore, this work did not require the models generate proofs the functions terminate and, because it evaluates models on publicly available problems, risks conflating memorized responses from training data with genuine capability. Our work requires proofs of termination and generates synthetic termination problems to avoid memorization.

We are aware of only one prior work by Kamath et al [21] on how LLMs do at *proving* functions terminate; however, the integration with the proving system is very minimal. Candidates are accepted or rejected based on a binary verification outcome, and failed attempts do not produce feedback. Instead, the model is repeatedly prompted until a correct answer is produced or time runs out (equivalent to the `nothing` configuration in our ablations). In contrast, our approach treats the theorem prover as a first-class component in the synthesis loop. Rather than using the prover solely as a post-hoc filter, we expose detailed prover feedback – such as failed subgoals, counterexamples, and intermediate proof obligations – to the model and leverage this information to guide subsequent generations. This enables iterative refinement driven by semantic proof state, allowing the model to adapt its hypotheses in response to specific proof failures.

Moreover, our work is distinct in two further ways. First, as discussed in the previous section, to prove a function terminates we use ACL2s' CCG and termination core algorithms to decompose it into simple loops, then we prompt the LLM to prove the termination of these, the LLM interacts with the theorem prover to do so, and finally the proofs are automatically combined to prove the whole function terminates. This inverts the typical setup where the LLM controls how a problem is decomposed, and shows how LLMs and existing

[2] At least one in 128 attempts yield a correct proof.

Algorithm 1 Theorem-Prover LLM Integration.

Require: function f
Require: time limit T
Require: generators: Model(), Prompt()
$M \leftarrow$ Model()
$t_0 \leftarrow$ Now()
$m \leftarrow M(\text{Prompt.Measure}(f))$
while Now() $- t_0 < T$ **do**
  if ContextWindowExhausted() **then**
    return `false`
  end if
  ▷ Admission
  $v \leftarrow$ ValidateMeasure(m)
  if $v \neq$ Ok **then**
    $m \leftarrow M(\text{Prompt.ValidationError}(m, f, v))$
    continue
  end if
  ▷ Verification
  if ACL2s(m, f) **then**
    return `true`
  end if
  ▷ Verification feedback
  $(kind, msg) \leftarrow$ DiagnoseFailure(m, f)
  if $kind =$ Counterexample **then**
    $m \leftarrow M(\text{Prompt.Counterexample}(m, f, msg))$
  else
    $m \leftarrow M(\text{Prompt.ProofFailed}(m, f, msg))$
  end if
end while
return `false`

theorem-proving automation can be integrated synergistically. Second, beyond implementing and evaluating an LLM theorem-prover integration, we run extensive ablations on what theorem prover feedback is actually useful to the model. Counterintuitively, we find that giving the model the full theorem-prover output (what a human user sees) performs worse than our more limited feedback configurations, and that what feedback is useful differs across models. We hope these results provide guidance for future authors of LLM–theorem-prover integrations.

3 Theorem-Prover Integration

In line with previous work [7, 13, 38, 41] our theorem-prover integration divides errors into two phases, admission and verification. After the LLM proposes a measure, we try and admit it with ACL2s and check the return type. If this fails, it is considered an admission error and the model is prompted to try again. If admission succeeds, we have the theorem prover try to prove the function's termination using the LLM's measure function. A proof attempt can fail because the measure is invalid, or because the measure is valid but the theorem prover failed to prove it decreases on recursive calls. For the former case, after proof failure we use ACL2s'

counterexample generation abilities [8,9] to find a counterexample to the measure working. If a counterexample is found, the feedback to the model is the counterexample, otherwise it is the theorem prover's diagnostic information about the proof failure. Algorithm 1 illustrates this process.

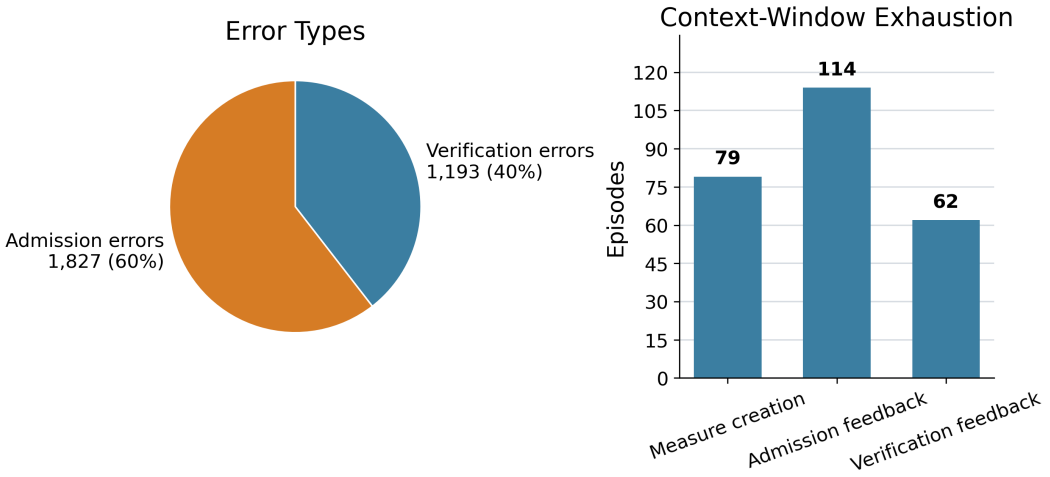

Figure 1 (Left) Distribution of error types during our ablation. (Right) Context-window exhaustion, a failure mode where the model outputs similar reasoning repeatedly until exhausting its context window, and when it occurred.

The left side of Figure 1 shows the distribution of admission and verification errors during our ablation experiments on `gpt-oss-120b`. Admission errors are more frequent than verification errors and tend to come from unbalanced parentheses and misuse of built-in functions. For example, in ACL2 the `floor` function takes two arguments and returns the floor of their division, as opposed to many other languages where `floor` takes a single argument. Non-standard design choices like this are a common reason for admission errors.

Our integration's feedback mechanism is parameterized by two variables: *history* and *phase*. If history is enabled, in addition to feedback about the most recently attempted measure, the model also receives a history of previous feedback. The phase is either `nothing`, `last attempt`, `admission`, or `verification`. When the phase is `nothing`, the model receives no feedback from the theorem prover. When the phase is `last attempt`, the model receives the last measure attempted as feedback. When the phase is `admission` or `verification`, the model receives admission errors and admission & verification errors respectively. For example, with history and `last attempt` the model sees all previously attempted measures as feedback; with history and `verification` the model sees each previously attempted measure along with the error that occurred when using it. Notice that the `nothing` phase is the same with and without history, as no feedback means the history of feedback is empty.

An important part of the feedback mechanism is that each error phase gives the model a strict superset of the information available in the previous one. That is, the information in `verification` is a superset of the information in `admission`, is a superset of `last attempt`, is a superset of `nothing`. This design is important for our ablation experiment (c.f. § 7) as it ensures a performance increase at a given phase is due only to the new information available at that phase.

4 Model Configuration

We run `gpt-oss-120b` and `Qwen3-32B` via vLLM with zero temperature on a single H200 GPU. `gpt-oss-120b` uses high reasoning effort. Despite zero temperature, the models' output is still non-deterministic [19] unless we fix a seed value in vLLM. Rather than doing our experiments with respect to 'model with fixed seed x', we use multiple samples with different seeds where appropriate in our experiments.

We chose these models for three reasons. First, they are some of the most powerful public models that can be run on a single H200 GPU. Second, querying a model via an API wouldn't allow us to get reliable timing information as the hardware can vary. Third, public APIs don't guarantee the model isn't quantized or otherwise changed throughout an experiment, using open-weight models ensures our work is reproducible.

While running `gpt-oss-120b` we noticed a failure mode where it enters a repetitive reasoning loop and exhausts its context window on hard problems. We call this context-window exhaustion. When this happens while trying to prove a function's termination, we consider the attempt failed, as there is no obvious way to continue with an exhausted context window. A small excerpt from the model's "thinking" during one of these episodes is shown below.

```
But we can try measure = y + 1 + (abs y) + (if y <= 0 then 3121 else 0).  But
we saw that fails for y = 1.  But we can try measure = y + 1 + (abs y) + (if
y <= 0 then 3122 else 0).  But we saw that fails for y = 1.  But we can try
measure = y + 1 + (abs y) + (if y <= 0 then 3123 else 0).  But we saw that
fails for y = 1.  But we can try measure = y + 1 + (abs y) + (if y <= 0 then
3124 else 0).  But we saw that fails for y = 1.  But we can try measure = y +
1 + (abs y) + (if y <= ...
```

The right side of Figure 1 shows the distribution of which phase context-window exhaustion occurred in during our ablation experiments.

5 Termination Problem Corpus

To evaluate LLMs' ability to prove functions terminate, we collect 42 interesting, terminating functions from a university-level Logic and Computation course [27] and past competitions for automatically proving function termination [30]. Problems from the termination competition are not written in ACL2, so these were converted by hand. `gpt-oss-120b` outperformed `Qwen3-32B`. In a configuration where the `gpt-oss-120b` receives counterexamples and admission errors, 70% of *Logic and Computation* problems were solved and 93% of termination competition problems were solved, surpassing ACL2s' built-in termination analysis [28] which solves 65% and 63% of problems respectively.

We'll highlight two problems `gpt-oss-120b` struggled with. ACL2s' built-in termination analysis is able to prove the termination of the first example automatically but fails on the second.

First, exercise 5.44 from the course textbook, below, was solved in one of twenty attempts by a configuration where the model receives admission errors and a history of attempts.

```
(definec f (x y acc :tl) :tl
  (cond ((^ (endp x) (endp y)) acc)
        ((endp x) (f x (rest y) (cons (first y) acc)))
        ((endp y) (f y x acc))
        (t (f x nil (f acc nil y)))))
```

The subtlety of this problem lies in the case where neither `x` nor `y` is empty: Here, the measure must decrease on both the inner and outer call to `f`, but because `f` has not been admitted, no reasoning can be done about the outer call's `acc` argument. In its one successful attempt of twenty the model proposes the measure below to handle this subtlety.

```
(definec m (x y acc :tl) :nat
  (let* ((lenx (len x))
         (leny (len y))
         (lenc (len acc))
         (cond-both
          (if (and (not (endp x)) (not (endp y))) 1 0))
         (cond-x-nonempty-y-empty
          (if (and (not (endp x)) (endp y)) 1 0)))
    (+ (* 2 lenx)
       (* 2 leny)
       (* 2 lenc cond-both)
       cond-x-nonempty-y-empty)))
```

Another problem at the edge of the model's abilities is `twn01.lisp` [26] which is solved twice in twenty attempts.

```
(definec f (a b :nat) :nat
  (if (and (< a b) (> a 0))
      (f (* 3 a) (* 2 b))
    0))
```

Intuitively, this function terminates because each recursive call reduces the ratio between `b` and `a` by $\frac{2}{3}$. A fact exploited by the model's measure function, below.

```
(definec m (a b :nat) :nat
  (if (or (<= a 0) (not (< a b)))
      0
    (floor (* 3 b) a)))
```

While this base corpus provides challenging termination problems, it is limited by public availability. This makes it possible solutions appear in the model's training data, and in turn responses are memorized. To address these limitations, we synthetically generate new termination problems from the base corpus.

6 Problem Synthesis

Many of the problems in our base corpus are publicly available, raising a concern that model responses are memorized. To ameliorate this, we generate synthetic termination problems from the base corpus and evaluate the model on these. We employ three techniques to generate hundreds of new problems by combining existing ones: *stack*, *race*, and *fuse*. These techniques are designed to increase the complexity of the control flow and variable dependencies, challenging the model's ability to reason about termination rather than relying on memorized structural patterns. To demonstrate these, we use the following two function definitions.

```
(definec append (x y :tl) :tl
  (if y (cons (car y) (append x (cdr y))) x))
```

```
(definec reverse (x :tl) :tl
  (and x (append (list (car x)) (reverse (cdr x)))))
```

Stacking `append` and `reverse` yields a combined function taking an additional boolean `flag` argument. This technique creates a function with disjoint termination conditions governed by a static control variable. If `flag` is true, the function mirrors the execution path of `append`; otherwise, it mirrors the path of `reverse`.

```
(definec stack (x0 y0 x1 :tl flag :bool) :tl
  (if flag
      (if y0
          (cons (car y0) (stack x0 (cdr y0) x1 flag))
        x0)
    (and x1
        (append (list (car x1)) (stack x0 y0 (cdr x1) flag)))))
```

Racing `append` and `reverse` yields a similar function, but the `flag` is toggled on every recursive call. This intertwines the execution paths of the original functions. Consequently, the active decreasing variable alternates with each step, requiring the termination analysis to track multiple variables across varying recursive depths.

```
(definec race (x0 y0 x1 :tl flag :bool) :tl
  (if flag
      (if y0
          (cons (car y0) (race x0 (cdr y0) x1 (not flag)))
        x0)
    (and x1
        (append (list (car x1)) (race x0 y0 (cdr x1) (not flag))))))
```

Fusing `append` and `reverse` introduces nested recursion. Here, the `flag` controls the inlining of `append` into the recursive step of `reverse`. By forcing the output of one recursive call to become the argument for another, this technique significantly complicates the data dependencies required to prove termination.

```
(definec fuse (x0 y0 x1 :tl flag :bool) :tl
  (if flag
      (if y0
          (cons (car y0) (fuse x0 (cdr y0) x1 flag))
        x0)
    (and x1
        (fuse (list (car x1)) (fuse x0 y0 (cdr x1) flag) x1 t))))
```

All of these methods can be extended to support more than two input functions. If the synthesis method is known, synthesizing a measure for the combined function is straightforward. If `m1` and `m2` are valid measures for the respective input functions, then the sum of these measures, `(+ (m1 x0 y0) (m2 x1))` using the variables from this example, serves as a valid measure for the synthetic function. We make use of this property in our decomposition experiment, § 9, to observe how the model's performance changes when this structural information is revealed.

7 Ablation Experiment

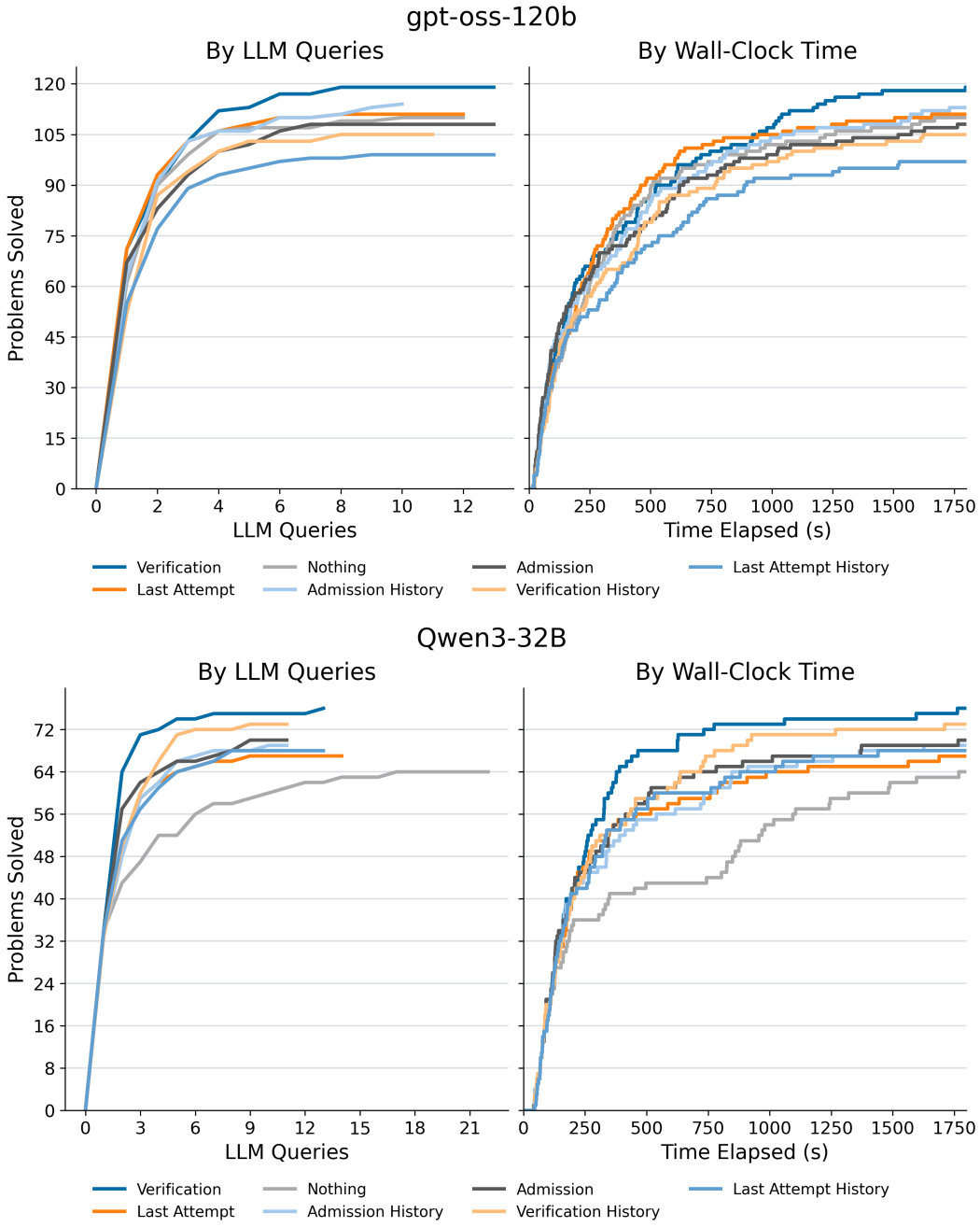

Figure 2 Comparison of different LLM theorem-prover integrations on the stack synthetic corpus with respect to number of LLM queries run and time elapsed. The model is given a thirty-minute time limit per problem.

To understand what theorem-prover feedback is useful to the LLMs we ran an ablation comparing all combinations of our integration's error and history parameters on the stacked synthetic problem set. The number of solved problems as a function of queries to the model and time are shown in Figure 2. For each problem the model was given thirty minutes

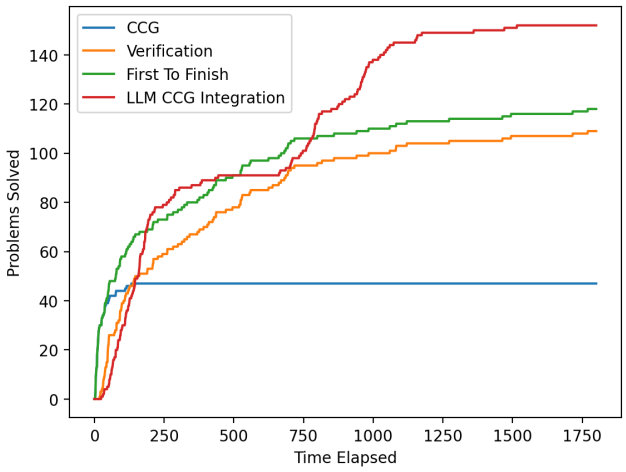

Figure 3 Comparison of our best-performing theorem-prover LLM integration (`gpt-oss-120b` with verification errors), ACL2s' built-in termination analysis [28], a combined configuration where the two methods are run in parallel and the first successful result returned, and a version where the model is integrated with ACL2s' built-in termination analysis to solve decomposed sub-problems.

to produce a solution. We compare performance as a function of queries because, by that metric, configurations that generate longer prompts with useful information aren't penalized by the model taking longer to process them. For example, measuring with respect to number of queries prevents an integration like `nothing` from gaining an advantage because it can sample the model more within the time limit.

By both measures, for both models, verification errors without history are the most effective. This is notable because the history configuration contains a superset of the information in the no-history configuration, yet performs worse. This suggests the history introduces noise that outweighs its utility for the model.

Also of note is the `nothing` configuration being top-four by both measures for `gpt-oss-120b`. This might explain why, as discussed in the introduction, leading models use limited theorem-prover feedback or explicitly train the model to respond to it. Support for this hypothesis can be found in the DeepSeek-Prover-V2 evaluations where, with a model that does not use theorem-prover error messages, on the minif2f dataset of formal Olympiad-level math problems, the model has a 55% success rate with one sample, but 73% success rate with 1024 samples: Evidently repeated sampling is a reasonably effective way to synthesize a proof.

8 Integration With ACL2s' Built-In Analysis

Figure 3 compares the best performing theorem-prover integration with ACL2s' built-in termination analysis [28] on the synthetic stack corpus and considers two integrations of the model with the built-in termination analysis. Like the LLM, the built-in analysis is given thirty minutes per problem.

The built-in analysis (blue line) is much faster initially, but its performance doesn't scale well with time. Interestingly, nine problems are solved that are not solved by the LLM. Though note that due to the model's non-determinism, the size and contents of this set might vary across runs. As a naive integration of the LLM and built-in analysis (green line) we consider a configuration where both are run in parallel and the first termination proof is taken as the answer, the difference between this and the verification configuration (orange line) is problems solved by built-in analysis but not the model.

Algorithm 2 Iterative core-guided termination via LLM-generated measures.

Require: function f
Require: time limit T
$t_0 \leftarrow \text{NOW}()$
$H \leftarrow \emptyset$ ▷ Accumulated measure hints
while $\text{NOW}() - t_0 < T$ **do**
  $(status, \mathcal{C}) \leftarrow \text{TERMINATIONALGORITHM}(f, H)$ ▷ $\mathcal{C}$ is a termination core.
  if $status = \text{PROVED}$ **then**
    return `true`
  end if
  if $\text{LLMINTEGRATION}(\mathcal{C}, T - (\text{NOW}() - t_0))$ **then**
    $m_c \leftarrow$ Measure generated by LLM
    $H \leftarrow H \cup \{m_c\}$
  else
    break
  end if
end while
return `false`

The best performing integration integrates the LLM with ACL2s' built-in termination analysis algorithms, an illustration of which is given by Algorithm 2. When ACL2s' termination analysis proves the termination of a function, it attempts to decompose the function into individual loops, called termination cores, and prove the termination of these [29]. For our stacked synthetic problems, this decomposition yields loops corresponding to their input functions. Using the `:consider-ccms` hint, the user of the theorem prover can provide measures which ACL2s' analysis will attempt to use for these loops. So, we can ask the model (with the verification configuration) to prove the termination of these loops (input functions) and provide the resulting measures as `:consider-ccms` hints. This (red line) is the best-performing theorem-prover LLM integration in our experiments, solving 39% more problems than the LLM and more than tripling the number solved by the built-in analysis.

While our implementation uses the ACL2s theorem prover, it is in principle usable with any theorem prover. Calling context graphs [28] and termination cores [29], the basis of ACL2s' analysis, are applicable to any functional language.

These experiments show that the proof automation provided by ACL2s is complementary to the LLM's abilities. First, while the model solves more problems in total, some problems remain out of its reach that are solvable by the built-in analysis. Second, the built-in analysis can help the model decompose an out-of-reach input function into smaller problems, which can be solved by the model. This synergistic configuration is the best-performing in our experiments; however, the model's failure to decompose these problems itself is interesting. We investigate this in our final set of experiments.

9 Decomposition Experiment

Because a measure for a synthetic function can be easily synthesized given measures for the functions it is derived from, our final experiment considers if `gpt-oss-120b` is able to decompose the synthetic functions into their constituent parts. We selected ten challenging problems from our ablation experiment, then evaluated the model twenty times on the problems the challenging problems were derived from, using the best-performing integration

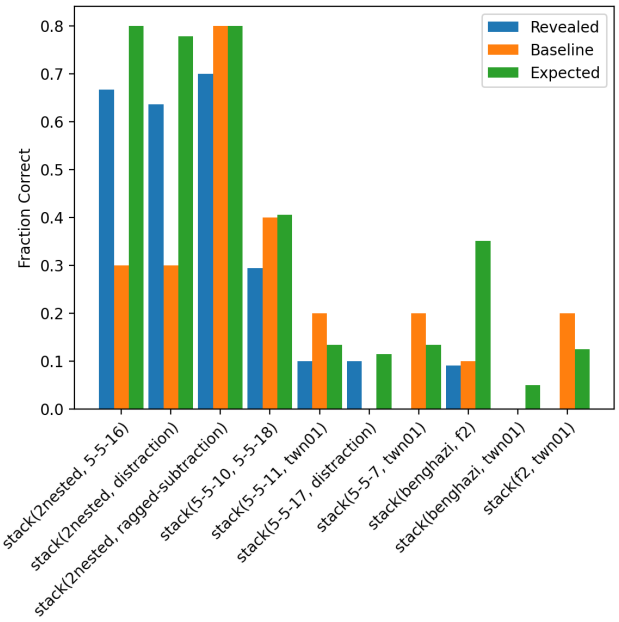

Figure 4 How well does `gpt-oss-120b` decompose problems? The expected performance is the probability that both inputs to the synthetic problem are solved, approximated by twenty samples. The revealed performance is the success rate when the synthesis method is revealed in the prompt; the baseline performance is the success rate when it is not.

strategy from our ablation. From this we estimate the probability the model solves each constituent problem. If p_1 and p_2 are the probability of solving two constituent problems, then if the model is perfectly decomposing the stacked version of the problems, the expected probability it solves the stacked problem is $p_1 \cdot p_2$; this is the "Expected" line in Figure 4. Figure 4 compares the model's success rate on twenty samples of each of the ten difficult problems with each problems expectation. The "Baseline" configuration uses the default prompt and the "Revealed" configuration adds information to the prompt explicitly telling the model how the stack synthesis method works and how to decompose the problem of generating a measure.

For seven of the ten problems, the model underperforms or meets expectation for both prompt variants, and there does not appear to be a major increase in performance when the synthesis method is revealed. For the problems where baseline outperforms expectation, we believe these are due to model randomness and sampling. To investigate, we also ran this experiment using the "Admission History" configuration and in that run the baseline underperformed expectation on every problem it overperforms in Figure 4's experiment. In aggregate, these results suggest the model is not perfectly decomposing the problem of generating a measure, even when instructions on how to do the decomposition are given in the prompt.

10 Conclusions and Future Work

We presented the first, to our knowledge, integration of an LLM and a symbolic termination-proving algorithm. By curating a base corpus of termination problems and introducing three synthesis techniques to generate novel functions, we avoided contamination from training data.

Across nine ablations for two different LLMs, we found that the form of theorem-prover feedback has a substantial impact on model performance. Our results show that feedback on proof failures improves performance, while a history of these errors can degrade performance. Furthermore, the performance of error-message configurations differed by model. This suggests that (1) while LLMs use error messages, excessive context can be detrimental, and (2) the efficacy of an error-message configuration can change depending on the model used. We also observed a failure mode for `gpt-oss-120b` where repetitive internal reasoning led to context exhaustion. Overall, both models outperformed ACL2s' built-in termination analysis.

Our decomposition experiment showed that `gpt-oss-120b` does not reliably decompose termination arguments, even when the structure of the synthetic function is revealed. With this in mind, we integrated the model with ACL2s' built-in termination analysis, which is good at decomposing a function into its individual loops. Having the model prove the termination of these inner loops and having the built-in analysis construct the final proof solved 39% more problems than the LLM alone and more than tripled the number solved by the built-in analysis on our benchmark. This demonstrates there can be a synergistic relationship between existing proof automation and new capabilities from an LLM.

We see many opportunities for integrating LLMs with theorem-proving algorithms. One obvious area to extend this work would be proving non-termination, using ACL2s to decompose the problem and asking the LLM to find a witness to non-termination. In a preliminary experiment, for example, GPT-5 [34] was able to generate a witness for the non-termination of `Zantema_15/ex07` despite no entries in Termination Competition 2025 being able to do so [44]. Furthermore, LLMs could be more deeply integrated with CCG to assist in proving one function call cannot result in a call to another, thereby decreasing the size of the graph. Finally, the model's difficulty decomposing problems warrants investigation; future work could inspect thinking traces or consider an agentic system where one model decomposes problems and others attempt to prove the termination of subparts.

11 Code Availability

Code for reproducing the experiments in this paper can be found at

`https://github.com/0xekez/fslatp`.

The code was handwritten with help from an LLM.

References

1 Tudor Achim, Alex Best, Kevin Der, Mathïs Fédérico, Sergei Gukov, Daniel Halpern-Leister, Kirsten Henningsgard, Yury Kudryashov, Alexander Meiburg, Martin Michelsen, Riley Patterson, Eric Rodriguez, Laura Scharff, Vikram Shanker, Vladmir Sicca, Hari Sowrirajan, Aidan Swope, Matyas Tamas, Vlad Tenev, Jonathan Thomm, Harold Williams, and Lawrence Wu. Aristotle: IMO-level Automated Theorem Proving, 2025. `doi:10.48550/arXiv.2510.01346`.

2 Agda Team. Sized Types. `https://agda.readthedocs.io/en/latest/language/sized-types.html`, 2025.

3 Anthropic. Claude sonnet 4.5, 2025. Large language model. URL: `https://www.anthropic.com/news/claude-sonnet-4-5`.

4 AxiomMathAI. Putnam, the world's hardest college-level math test, ended yesterday ... AxiomProver solved 9/12 problems in Lean autonomously, scoring at top of 4000 participants. Tweet, 2025. Posted on X (formerly Twitter), Dec 7, 2025. URL: `https://x.com/axiommathai/status/1997767850279440715`.

5 Dirk Beyer. Competition on software verification (sv-comp). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, pages 504–524. Springer, 2012. doi:10.1007/978-3-642-28756-5_38.

6 Robert S. Boyer and J Strother Moore. *A Computational Logic*. Academic Press, New York, 1979. Describes the NQTHM (Boyer–Moore) theorem prover.

7 Sergiu Bursuc, Theodore Ehrenborg, Shaowei Lin, Lacramioara Astefanoaei, Ionel Emilian Chiosa, Jure Kukovec, Alok Singh, Oliver Butterley, Adem Bizid, Quinn Dougherty, Miranda Zhao, Max Tan, and Max Tegmark. A Benchmark for Vericoding: Formally Verified Program Synthesis, 2025. doi:10.48550/arXiv.2509.22908.

8 Harsh Raju Chamarthi, Peter C. Dillinger, Matt Kaufmann, and Panagiotis Manolios. Integrating Testing and Interactive Theorem Proving. In David S. Hardin and Julien Schmaltz, editors, *Proceedings 10th International Workshop on the ACL2 Theorem Prover and its Applications*, volume 70 of *EPTCS*, pages 4–19, 2011. doi:10.4204/EPTCS.70.1.

9 Harsh Raju Chamarthi and Panagiotis Manolios. Automated Specification Analysis Using an Interactive Theorem Prover. In W. A. Hunt Jr. and S. D. Johnson, editors, *Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2011)*, pages 92–99, Austin, TX, USA, 2011.

10 Luoxin Chen, Jinming Gu, Liankai Huang, Wenhao Huang, Zhicheng Jiang, Allan Jie, Xiaoran Jin, Xing Jin, Chenggang Li, Kaijing Ma, Cheng Ren, Jiawei Shen, Wenlei Shi, Tong Sun, He Sun, Jiahui Wang, Siran Wang, Zhihong Wang, Chenrui Wei, Shufa Wei, Yonghui Wu, Yuchen Wu, Yihang Xia, Huajian Xin, Fan Yang, Huaiyuan Ying, Hongyi Yuan, Zheng Yuan, Tianyang Zhan, Chi Zhang, Yue Zhang, Ge Zhang, Tianyun Zhao, Jianqiu Zhao, Yichi Zhou, and Thomas Hanwen Zhu. Seed-Prover: Deep and Broad Reasoning for Automated Theorem Proving, 2025. doi:10.48550/arXiv.2507.23726.

11 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: Beyond Safety. In *Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 415–418. Springer, 2006. doi:10.1007/11817963_37.

12 Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2–3):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

13 Marcos Cramer and Lucian McIntyre. Verifying LLM-Generated Code in the Context of Software Verification with Ada/SPARK, 2025. doi:10.48550/arXiv.2502.07728.

14 Leonardo de Moura and Sebastian Ullrich. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction – CADE 28*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

15 Peter C. Dillinger, Panagiotis Manolios, Daron Vroon, and J Strother Moore. ACL2s: "The ACL2 Sedan". In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007, Companion Volume*, pages 59–60. IEEE Computer Society, 2007. doi:10.1109/ICSECOMPANION.2007.14.

16 Kefan Dong and Tengyu Ma. STP: Self-play LLM Theorem Provers with Iterative Conjecturing and Proving. *CoRR*, abs/2502.00212, 2025. doi:10.48550/arXiv.2502.00212.

17 Jürgen Giesl, Carsten Aschermann, Marc Brockschmidt, Fabian Emmes, Falko Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephan Swiderski, and René Thiemann. Analyzing Program Termination and Complexity Automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

18 Anthony Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990. doi:10.1109/52.57819.

19 Horace He and Thinking Machines Lab. Defeating Nondeterminism in LLM Inference. Thinking Machines Lab: Connectionism, 2025. doi:10.64434/tml.20250910.

20 Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, and Ting Liu. A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions. *ACM Trans. Inf. Syst.*, 43(2):42:1–42:55, 2025. doi:10.1145/3703155.

21 Adharsh Kamath, Nausheen Mohammed, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. Leveraging llms for program verification. In Nina Narodytska and Philipp Rümmer, editors, *Proceedings of the 24th Conference on Formal Methods in Computer-Aided Design – FMCAD 2024*, pages 107–118. TU Wien Academic Press, 2024. doi:10.34727/2024/isbn.978-3-85448-065-5_16.

22 Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Springer, 2000. doi:10.1007/978-1-4757-3188-0.

23 Angelo Kyrilov and David C. Noelle. Using Automated Theorem Provers to Teach Knowledge Representation in First-Order Logic. *CoRR*, abs/1507.03670, 2015. arXiv:1507.03670.

24 Lean Prover Community. Recursive Definitions (Lean 4 Reference Manual). https://lean-lang.org/doc/reference/4.19.0/Definitions/Recursive-Definitions/, 2025.

25 Yong Lin, Shange Tang, Bohan Lyu, Jiayun Wu, Hongzhou Lin, Kaiyu Yang, Jia Li, Mengzhou Xia, Danqi Chen, Sanjeev Arora, and Chi Jin. Goedel-Prover: A Frontier Model for Open-Source Automated Theorem Proving. *CoRR*, abs/2502.07640, 2025. doi:10.48550/arXiv.2502.07640.

26 Nils Lommen, Fabian Meyer, and Jürgen Giesl. Automatic Complexity Analysis of Integer Programs via Triangular Weakly Non-Linear Loops. In Jasmin Blanchette, Laura Kovács, and Dirk Pattinson, editors, *Automated Reasoning (IJCAR 2022)*, volume 13385 of *Lecture Notes in Computer Science*, pages 734–754. Springer, 2022. doi:10.1007/978-3-031-10769-6_43.

27 Panagiotis Manolios. Reasoning About Programs. Course notes, Computer-Aided Reasoning, Northeastern University, 2026. URL: https://www.ccs.neu.edu/home/pete/courses/Computer-Aided-Reasoning/2026-Spring/ReasoningAboutPrograms.pdf.

28 Panagiotis Manolios and Daron Vroon. Termination Analysis with Calling Context Graphs. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification (CAV 2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 401–414. Springer, 2006. doi:10.1007/11817963_36.

29 Panagiotis Manolios and Daron Vroon. Interactive Termination Proofs Using Termination Cores. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving (ITP)*, volume 6172 of *Lecture Notes in Computer Science*, pages 355–370. Springer, 2010. doi:10.1007/978-3-642-14052-5_25.

30 Claude Marché and Hans Zantema. The Termination Competition. In Franz Baader, editor, *Term Rewriting and Applications (RTA 2007)*, volume 4533 of *Lecture Notes in Computer Science*, pages 303–313. Springer, 2007. doi:10.1007/978-3-540-73449-9_23.

31 Zeke Medley and Panagiotis Manolios. 0xekez/fslatp. Software, swhId: swh:1:dir:26cbefe528532dda47d04b7198cca383281e5a72 (visited on 2026-07-14). URL: https://github.com/0xekez/fslatp, doi:10.4230/artifacts.27134.

32 Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models. Apple Machine Learning Research (paper page), 2024. URL: https://machinelearning.apple.com/research/gsm-symbolic.

33 Iman Mirzadeh, Keivan Alizadeh, Hooman Shahrokhi, Oncel Tuzel, Samy Bengio, and Mehrdad Farajtabar. GSM-Symbolic: Understanding the Limitations of Mathematical Reasoning in Large Language Models. *arXiv preprint*, 2024. doi:10.48550/arXiv.2410.05229.

34 OpenAI. Gpt-5, 2025. Large language model. URL: https://openai.com/gpt-5.

35 OpenAI. gpt-oss-120b & gpt-oss-20b Model Card, 2025. doi:10.48550/arXiv.2508.10925.

36 Christine Paulin-Mohring. Inductive Definitions in the System Coq: Rules and Properties. In *Typed Lambda Calculi and Applications (TLCA 1993)*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993. doi:10.1007/BFb0037116.

37 Stanislas Polu and Ilya Sutskever. Generative Language Modeling for Automated Theorem Proving. *CoRR*, abs/2009.03393, 2020. arXiv:2009.03393.

38 Jianxing Qin, Alexander Du, Danfeng Zhang, Matthew Lentz, and Danyang Zhuo. Can Large Language Models Verify System Software? A Case Study Using FSCQ as a Benchmark. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '25)*, pages 34–41, New York, NY, USA, 2025. ACM. doi:10.1145/3713082.3730382.

39 Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing Formal Mathematical Reasoning via Reinforcement Learning for Subgoal Decomposition. *CoRR*, abs/2504.21801, 2025. doi:10.48550/arXiv.2504.21801.

40 The Rocq Development Team. *The Rocq Prover Reference Manual*, 2024. URL: https://rocq-prover.org/refman.

41 Aleksandr Shefer, Igor Engel, Stanislav Alekseev, Daniil Berezun, Ekaterina Verbitskaia, and Anton Podkopaev. Can LLMs Enable Verification in Mainstream Programming?, 2025. doi:10.48550/arXiv.2503.14183.

42 Oren Sultan, Jordi Armengol-Estapé, Pascal Kesseli, Julien Vanegue, Dafna Shahaf, Yossi Adi, and Peter O'Hearn. Llms versus the halting problem: Revisiting program termination prediction. *arXiv preprint v3*, 2026. doi:10.48550/arXiv.2601.18987.

43 FAIR CodeGen Team, Jade Copet, Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, Kunhao Zheng, Jordi Armengol-Estapé, Pedram Bashiri, Maximilian Beck, Pierre Chambon, Abhishek Charnalia, Chris Cummins, Juliette Decugis, Zacharias V. Fisches, François Fleuret, Fabian Gloeckle, Alex Gu, Michael Hassid, Daniel Haziza, Badr Youbi Idrissi, Christian Keller, Rahul Kindi, Hugh Leather, Gallil Maimon, Aram Markosyan, Francisco Massa, Pierre-Emmanuel Mazaré, Vegard Mella, Naila Murray, Keyur Muzumdar, Peter O'Hearn, Matteo Pagliardini, Dmitrii Pedchenko, Tal Remez, Volker Seeker, Marco Selvi, Oren Sultan, Sida Wang, Luca Wehrstedt, Ori Yoran, Lingming Zhang, Taco Cohen, Yossi Adi, and Gabriel Synnaeve. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint*, 2025. doi:10.48550/arXiv.2510.02387.

44 Termination Competition. Termination competition 2025: Trs standard. https://termcomp.github.io/Y2025/TRS_Standard.html, 2025. Accessed: 2026-05-21.

45 Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An In-Context Learning Agent for Formal Theorem-Proving, 2023. doi:10.48550/arXiv.2310.04353.

46 Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-Prover Preview: Towards Large Formal Reasoning Models with Reinforcement Learning, 2025. doi:10.48550/arXiv.2504.11354.

47 Wolfgang Windsteiger. Automated Theorem Proving in the Classroom. In Predrag Janicic and Zoltán Kovács, editors, *Proceedings of the 13th International Conference on Automated Deduction in Geometry, ADG 2021, Hagenberg, Austria/virtual, September 15-17, 2021*, volume 352 of *EPTCS*, pages 54–63, 2021. doi:10.4204/EPTCS.352.6.

48 Huajian Xin, Daya Guo, Zhihong Shao, Z. Z. Ren, Qihao Zhu, Bo Liu, Chong Ruan, Wenda Li, and Xiaodan Liang. DeepSeek-Prover: Advancing Theorem Proving in LLMs through Large-Scale Synthetic Data, 2024. doi:10.48550/arXiv.2405.14333.

49 Huajian Xin, Z. Z. Ren, Junxiao Song, Zhihong Shao, Wanjia Zhao, Haocheng Wang, Bo Liu, Liyue Zhang, Xuan Lu, Qiushi Du, Wenjun Gao, Haowei Zhang, Qihao Zhu, Dejian Yang, Zhibin Gou, Z. F. Wu, Fuli Luo, and Chong Ruan. DeepSeek-Prover-V1.5: Harnessing Proof Assistant Feedback for Reinforcement Learning and Monte-Carlo Tree Search. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025. URL: https://openreview.net/forum?id=I4YAIwrsXa.

50 A. Yang et al. Qwen3: Large Language Models with Hybrid Thinking and Non-Thinking Modes. *arXiv preprint*, 2025. arXiv:2505.09388.

51 Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking Verifiable Code Generation. *arXiv preprint*, 2025. arXiv:2505.23135, doi:10.48550/arXiv.2505.23135.

52 Kunhao Zheng, Jesse Michael Han, and Stanislas Polu. MiniF2F: a cross-system benchmark for formal Olympiad-level mathematics. *arXiv preprint*, 2021. arXiv:2109.00110.