Solving P-99 at the age of LLM: examples and pedagogical considerations*

Fred Mesnard[1], Thierry Marianne[1], Etienne Payet[1] and Wim Vanhoof[2]

[1]LIM, Université de La Réunion, France
[2]Université de Namur, Belgium

Abstract

Ninety-Nine Prolog Problems (P-99) is a famous set of Prolog exercises. We solved the first thirty three *just by prompting an LLM* (Large Language Model). We used Claude from Anthropic. By "solved" we mean: generate the Prolog code and a test file, run the tests and check whether they pass, then formally prove types, groundness, termination, uniqueness, existence and also sometimes functional correctness with LPTP (Logic Program Theorem prover). Hence our approach is an experiment in *vibe-coding/vericoding* of P-99. It is a *vibe-coding* experiment because we started from informal specifications written in English and let Claude generate the Prolog code. It also fits within *vericoding* because the LLM proved *reliability guarantees* on the generated Prolog code. Claude wrote 58 logic procedures, 508 tests, 257 lemmas for a total of 11800 proof lines. We manually checked each file generated by the LLM. We checked the Prolog code, ran the tests, examined the logical statements generated by Claude and proof-checked Claude's proofs with LPTP.

This paper describes this experiment and provides the main details so that it can be reproduced by the interested reader. It also includes some pedagogical considerations for teaching *certified logic programming* with the help of an LLM.

Keywords

Logic Programming, Prolog, Verification, LPTP, LLM

1. Introduction

The recent arrival and rapid adoption of LLMs (Large Language Models) and their growing aptitude at writing and/or dealing with code has triggered different AI assisted programming practices or methodologies. In this paper, we start by presenting such a methodology, applied to P-99. Ninety-Nine Prolog Problems (P-99) is a famous set of Prolog exercises that have been translated to many other programming languages, including Erlang, Haskell, Lisp, Picat and Rust. There is also an Active Logic Document [1] for P-99 running Ciao-Prolog in any modern web browser[1]. The oldest copy of P-99 we have dates back to more than 25 years. These exercises were written by Werner Hett from the Bern University of Applied Sciences, Switzerland. The original site has been offline for many years but there are a few copies floating on the Internet (*e.g.*, here[2]). Some exercises are missing (P29, P30, P42-P45, P51-P53, P74-P79) and a few of them have extensions (P61A, P62B, P70B, P70C). All in all, there are 88 exercises.

In this work, we start by presenting a case study using the combination of an LLM and LPTP (Logic Program Theorem Prover) [2] to solve some of the exercises of P-99. We choose Claude from Anthropic and do the experiment in *code mode* with the Opus 4.6 model released in February 2026. We solve the first thirty three exercises (P01-P28 + P31-P35 = 33/88 = 37.5%) just by *prompting Claude*. By "solving an exercise" we mean: generate the Prolog code and a test file, run the tests and check whether they pass, then formally prove types, groundness, termination, uniqueness, existence and also

Fourth Prolog Education Workshop, PEG 2026

*This work is partly funded by the European Union under the INTERREG VI "Mise en réseau" program. It shares content with a technical paper submitted to ICLP 2026. The main difference is that we replace a section describing a Model Context Protocol server by a section about pedagogical issues.

frederic.mesnard@univ-reunion.fr (F. Mesnard); thierry.marianne@univ-reunion.fr (T. Marianne); etienne.payet@univ-reunion.fr (E. Payet); wim.vanhoof@unamur.be (W. Vanhoof)

[1]https://cliplab.org/logalg/doc/99problemsALD.html/

[2]https://www.ic.unicamp.br/~meidanis/courses/mc336/2009s2/prolog/problemas/

sometimes functional correctness with LPTP. In other words, our case study relates an experiment in *vibe-coding/vericoding* of P-99. It is a *vibe-coding* [3] experiment because we start from informal specifications written in English for students and let Claude generate the Prolog code. It also fits within a *vericoding* approach [4] because the LLM proved *reliability guarantees* on the generated Prolog code in a formal language that can be proof-checked. We manually checked each of Claude's outputs. For functional correctness properties (informally: properties stating that the code computes what it is supposed to), we almost always provided hints to Claude to express the logical statements we were interested in.

This paper describes this experiment and provides the main details so that it can be reproduced by the interested reader. It also includes some pedagogical considerations for teaching such an approach to LP code generation and verification. It is organized as follows. Section 2 defines the experimental framework. Sections 3 and 4 describe how to instruct Claude about what it needs to know. Section 5 gives an overview of the experiment. Section 6 presents three examples with an emphasis on functional correctness properties. Section 7 discusses pedagogical considerations. Finally, Section 8 reviews some related work and Section 9 concludes.

2. Goal

The first problem of P-99 is given as follows:

```
P01 (*) Find the last element of a list.
Example:
?- my_last(X,[a,b,c,d]).
X = d
```

This is an informal specification written in English. In general, specifications include a few example queries that indicate the predicate name, arity and expected usage. The `(*)` indicates the difficulty of the exercise, ranging from one to three stars.

We want the LLM to propose the corresponding Prolog code together with some tests. We want proven lemmas insuring types, groundness, termination and functional properties such as links with library predicates or previously solved exercises. We want to repeat this exercise for the next 32 problems.

More precisely, the programming language we instruct Claude to use is pure Prolog with equality `=/2` on finite trees, negation as failure, the if/then/else construct and *no* builtins. Natural numbers are represented using Peano notation (*e.g.*, 2 is noted as `s(s(0))`). The Prolog engine is assumed to run unification *with* occurs-check. SWI-Prolog has a special flag for this mode. Claude adds the corresponding directive[3] at the beginning of each test file. The specification language and proof-checker is LPTP [5, 6, 2], see also the appendix and the companion paper [7].

3. Claude setup

Let us download the GitHub repository[4] accompanying this paper. We get the `LPTP-LLM-P99-main` directory, which contains the files `CLAUDE.md` (see the next section), `P-99.html`, the P-99 Prolog problems in HTML format and the file `lptp-reference.md`. In this last file, Claude *itself* maintains tips and lessons learnt about the project, such as how to run LPTP and Prolog locally, common pitfalls and proof patterns.

For each `Pxx` from `P99`, we want Claude to create a directory `P99/Pxx`, with its own `CLAUDE.md` file describing the problem, its solution `pxx.pl`, its tests `pxx_test.pl`, the properties and their proofs `pxx.pr`, and a report `pxx-experiment-report.md` describing the work done. The file `pxx.gr` is a

[3] `:- set_prolog_flag(occurs_check,true).`
[4] https://github.com/FredMesnard/LPTP-LLM-P99

representation created by Claude of the Prolog file pxx.pl, to be used by LPTP. Table 1 displays the structure of the initial directory.

Table 1
Layout of the GitHub repository accompanying the paper

Directory	Content
P99/	CLAUDE.md, P-99.html, lptp-reference.md
P99/P01/	CLAUDE.md, p01-experiment-report.md, p01.gr, p01.pl, p01.pr, p01_test.pl

The next section is an exact copy of the contents of the file P99/CLAUDE.md. It contains all the instructions for Claude to solve a P-99 exercise. It has to be adapted to the local configuration. Once all the infrastructure is ready, the following prompt can be used for solving P02: *Read CLAUDE.md, lptp-reference.md, and solve the P02 exercise.*

4. CLAUDE.md

4.1. Project

Formal verification of the 99 Prolog Problems using LPTP (Logic Program Theorem Prover) v1.06. The goal is to provide Prolog code, a test file, and LPTP proofs of properties for some of the 99 Prolog Problems.

4.2. Reference

Read /Users/fred/Desktop/P99/lptp-reference.md before writing or debugging any .pr file. Update this reference if a new proof tip has been found.

4.3. Structure

The problems are listed in P-99.html. Each problem lives in a Pxx/ directory containing:

- pXX.pl — Prolog program
- pXX.gr — ground representation (variable-free clause encoding for LPTP)
- pXX.pr — proof file (lemmas, theorems)
- pXX_test.pl — SWI-Prolog test file (see Testing below)
- pXX-experiment-report.md — experiment report
- CLAUDE.md — problem specification

4.4. Conventions

- **Prolog** strictly ISO-compatible, = (equality, *i.e.*, finite-tree unification), \+ (negation as failure), and (... -> ... ; ...) (the if-then-else construct) are allowed, but no cut, no other built-in.
- Use predicates from the LPTP lib when available, as it will help for the proofs.
- Peano naturals: 0, s(0), s(s(0)), ... using the LPTP nat library.
- Hierarchical lemma names: predicate:property or predicate:property:variant.
- The predicates defined in pXX.pl and pXX.gr must correspond exactly.
- Copy .gr and .pr to /Users/fred/lptp/tmp/ before verification.
- Command: cd /Users/fred/lptp && printf "io__exec_file('tmp/pXX.pr').\nhalt.\n" | bin/lptp 2>&1
- If pXX.pr uses lemmas from another problem (*e.g.*, P35 imports P31), declare the dependency via :- needs_gr / :- needs_thm and document it in the local CLAUDE.md.

4.5. Testing

For each problem Pxx, create a test file `pXX_test.pl` that:

- Starts with `:- set_prolog_flag(occurs_check,true).` as the very first line.
- Loads the program via `:- [pXX].`
- Includes helpers for Peano conversion (`to_peano/2`, `from_peano/2`, etc.) when needed.
- Tests each predicate defined in `pXX.pl` with representative cases: typical inputs, edge cases (empty list, 0, single element), and expected failures.
- Tests each semantic property proved in `pXX.pr` (types, uniqueness, ordered, product, etc.) as a runtime check.
- Prints `OK` or `FAIL` for each test case.
- Ends with `:- halt.`
- Runs with: `/Applications/SWI-Prolog.app/Contents/MacOS/swipl pXX_test.pl`

4.6. Properties to verify systematically

1. **Types** (`pred:types`) — type preservation (list, nat, etc.)
2. **Groundness** (`pred:ground`) — ground inputs ⇒ ground outputs
3. **Termination** (`pred:termination`) — termination under preconditions
4. **Uniqueness** (`pred:uniqueness`) — determinism of the result
5. **Existence** (`pred:existence`) — existence of the result
6. **Functional Correctness** — any link with library predicates or previously solved exercises

4.7. Experiment Report

Each `pXX-experiment-report.md` should contain:

- **Problem statement** — what the predicate does
- **Prolog code** — summary of the predicates defined
- **Properties proved** — list of lemmas with their statement and proof technique (completion, structural induction, strengthened induction, algebraic, etc.)
- **Statistics** — line counts (`.pl`, `.gr`, `.pr`), number of lemmas, proof-to-code ratio
- **Difficulties and lessons learned** — LPTP pitfalls encountered, proof strategies that worked

4.8. Language

- Code, lemma names, and LPTP proofs: English.
- Experiment reports and documentation: English.
- Conversation with the user: French (the user's preferred language).

5. Overview of the experiment

Together with Claude, we solved the first 33 Prolog exercises. Understanding the specification, writing the Prolog code and the test file takes a few minutes. We noticed that from time to time, a *context compaction* is followed by *this session is being continued from a previous conversation that ran out of context.* It can be useful to tell Claude to reread the `CLAUDE.md` files as Claude may have forgotten some guidelines. Switching to the verification part, the process is much slower. For functional properties, we had to give hints to Claude. For instance, for P01 and P02, we asked: *what is the connection with append/3?* Sometimes we had to fully formulate the properties in natural language (see Section 6.3). Solving one exercise takes Claude between 15 minutes (*e.g.*, P01) to several hours (*e.g.*, P35).

During this experiment, Claude created 58 logic procedures, with a total of 112 Prolog clauses, 150 lines of code and about 500 runtime tests. It did make use of negation by failure but made no use of the if/then/else construct. As we explicitly asked for pure Prolog code, the code generated by Claude is often quite different from the Prolog solutions one may find on the Internet (see *e.g.*, Section 6.2 and Section 6.3). Claude proved 257 lemmas and wrote about 11800 lines of proof. It had no problem in using what is already available in the LPTP library, code and lemmas. While proving its lemmas, it

found various proof techniques which are documented in the `lptp-experiment.md` file. An example of such a proof technique is the following: sometimes we have to strengthen the property to be proved so that we can do the inductive proof, and then weaken the property to get the original one. The human readable `lptp-experiment.md` file is maintained by Claude and is included in the GitHub repository accompanying this paper to speed up another run of this experiment.

A word on reproducibility. Since LLMs are stochastic, replaying our prompts will not, in general, regenerate the same code, tests and proofs: a given informal specification may admit several correct solutions, and the LLM may produce a different one, or even an incorrect one. So by *reproducible* we mean two things: the methodology of Section 3 can be replayed on each exercise, and the generated artefacts are independently *re-checkable* — anyone can rerun the tests and replay the LPTP proofs. This is where the combination LLM+LPTP pays off: trustworthiness does not rest on the LLM regenerating the same code, but on the fact that whatever code is produced comes with machine-checked guarantees.

6. Selected examples

6.1. P01: the last element of a list

The problem statement of exercise P01 was given in Section 2. Claude produces the following Prolog code (naive solution):

```
my_last(X, [X]).
my_last(X, [_|L]) :- my_last(X, L).
```

It also infers and proves 10 properties, see Table 2. We simplify the LPTP syntax for readability. The full statements and their LPTP-checked proofs are listed in the `p01.pr` file. The `p01_test.pl` file contains 21 tests, both for the source code and for the first 7 properties. The last three properties are Claude's answer to our question: *what is the connection with append/3?* Actually, properties 8 and 10 constitute a characterization of `my_last/2` w.r.t. `append/3`.

Table 2
Properties proved for `my_last/2`

#	Property	Statement
1	Termination	$\forall x, l.\ \texttt{list}(l) \Rightarrow \textbf{terminates}\ \texttt{my\_last}(x, l)$
2	Type preservation	$\forall x, l.\ \texttt{my\_last}(x, l) \Rightarrow \texttt{list}(l)$
3	Groundness	$\forall x, l.\ \texttt{my\_last}(x, l) \wedge \textbf{gr}(l) \Rightarrow \textbf{gr}(x)$
4	Membership	$\forall x, l.\ \texttt{my\_last}(x, l) \Rightarrow \texttt{member}(x, l)$
5	Uniqueness	$\forall x, l.\ \texttt{my\_last}(x, l) \Rightarrow (\forall y.\ \texttt{my\_last}(y, l) \Rightarrow x = y)$
6	Existence (aux.)	$\forall z, l.\ \texttt{list}(l) \Rightarrow \exists x.\ \texttt{my\_last}(x, [z\|l])$
7	Existence	$\forall l.\ \texttt{list}(l) \wedge l \neq [] \Rightarrow \exists x.\ \texttt{my\_last}(x, l)$
8	Forward append	$\forall x, l.\ \texttt{my\_last}(x, l) \Rightarrow \exists l_1.\ \texttt{append}(l_1, [x], l)$
9	Converse append (aux.)	$\forall x, l_1, l.\ \texttt{list}(l_1) \wedge \texttt{append}(l_1, [x], l) \Rightarrow \texttt{my\_last}(x, l)$
10	Converse append	$\forall x, l_1, l.\ \texttt{append}(l_1, [x], l) \Rightarrow \texttt{my\_last}(x, l)$

6.2. P31: a primality test

P31 asks for a primality test.

```
P31 (**) Determine whether a given integer number is prime.
Example:
?- is_prime(7).
Yes
```

Claude generates the following Prolog code:

```
% divides(D, N) - D divides N (by repeated subtraction, D > 0).
divides(D, D).
divides(D, N) :- plus(D, M, N), divides(D, M).

% no_factor(K, D, N) - none of D, D+1, ..., D+K-1 divides N.
%   K is a countdown: the number of candidates left to check.
no_factor(0, _, _).
no_factor(s(K), D, N) :- \+ divides(D, N), no_factor(K, s(D), N).

% is_prime(N) - N is prime (N >= 2 and no factor from 2 to N-1).
%   For N = s(s(X)), check X candidates starting from 2.
is_prime(s(s(X))) :- no_factor(X, s(s(0)), s(s(X))).
```

In Table 3 we present two properties stating soundness and completeness of the implementation. We asked for such properties and gave the idea of their formalization in natural language. Then Claude was able to formalize and prove them.

Table 3
Two properties proved for `is_prime/1`

#	Property	Statement
7	Soundness of `is_prime`	$\forall n.\ \texttt{nat}(n) \wedge \texttt{is\_prime}(n) \Rightarrow$ $\exists x.\ n = s(s(x)) \wedge (\forall d_1.\ s(0) \leq d_1 \wedge d_1 < s(x) \Rightarrow$ **fails** $\texttt{divides}(s(d_1), n))$
9	Completeness of `is_prime`	$\forall n, x.\ \texttt{nat}(n) \wedge n = s(s(x)) \wedge$ $(\forall d_1.\ \ s(0) \leq d_1 \wedge d_1 < s(x) \Rightarrow$ **fails** $\texttt{divides}(s(d_1), n))$ $\Rightarrow \texttt{is\_prime}(n)$

6.3. P35: decomposition in prime numbers

P35 asks for a decomposition of a natural number in prime factors.

```
P35 (**) Determine the prime factors of a given positive integer.
Construct a flat list containing the prime factors in ascending order.
Example:
?- prime_factors(315, L).
L = [3,3,5,7]
```

Here is the Prolog code generated by Claude:

```
% quot(D, N, Q) - quotient Q = N/D (assumes D divides N, D > 0).
quot(D, D, s(0)).
quot(D, N, s(Q)) :- plus(D, M, N), quot(D, M, Q).

% smallest_factor(N, D, K, F) - smallest factor F of N starting from
%   candidate D, with countdown K for termination.
%   K counts how many candidates remain to try after D.
smallest_factor(N, D, K, D) :- divides(D, N).
smallest_factor(N, D, s(K), F) :- \+ divides(D, N),
    smallest_factor(N, s(D), K, F).
```

```
% prime_factors(N, L) - L is the list of prime factors of N (>= 1)
%   in ascending order, with repetitions.
%   N = 0 fails (not a positive integer).
prime_factors(s(0), []).
prime_factors(s(s(X)), [F|L]) :-
    smallest_factor(s(s(X)), s(s(0)), X, F),
    quot(F, s(s(X)), Q),
    prime_factors(Q, L).
```

Table 4 lists the properties of this program. Claude started with termination and type properties. For stating and proving properties 23 to 30 that we explicitly asked in natural language, Claude reused code and lemmas from the `nat` library (*e.g.*, `nat/1`, `plus/3`, `times/3`, `@=< /2`), derived and proved new lemmas for P31 (`divides/3`), P35 (`quot/3`, `smallest_factor/4`, `prime_factors/2`) and added Prolog code (`ordered/1`, `product/2`):

```
% ordered(L) - L is sorted in ascending order (using @=<).
ordered([]).
ordered([_]).
ordered([X,Y|L]) :- X @=< Y, ordered([Y|L]).

% product(L, P) - P is the product of elements of L.
product([], s(0)).
product([X|L], P) :- product(L, P1), times(X, P1, P).
```

We note that the product of the natural numbers contained in the empty list is 1. Let us focus on the last properties. Assuming n is a Peano integer and $prime\_factors(n, l)$ succeeds, here is what Claude proved:

- property 23 (or 24): the resulting list l is a list of Peano integers;
- property 26: the product of the elements of l is n;
- property 27: l is in ascending order;
- property 28: each element of l is a prime number;
- property 30: l is unique.

We also get a sufficient condition for the existence of a solution:

- property 29: if n is strictly greater than 0, there exists l such that $prime\_factors(n, l)$ succeeds.

All together, these results constitute a logic-programming-based proof of the *prime factorization theorem*: "every integer greater than 1 is either prime or can be represented uniquely as a product of prime numbers, up to the order of the factors"[5]. On the one hand, the Prolog code effectively constructs the ordered list l of prime factors of a strictly positive natural number n. On the other hand, the proofs certify that if n is strictly greater than 0 then l always exists, is unique and correct.

7. Pedagogical considerations

We present some initial debatable ideas for educating students to this way of constructing certified logic programs, without claiming for originality. We believe that for software engineers, FOL (first order logic) is a must-know language at least for understanding and expressing specifications. We distinguish two entry points for teaching certified LP at the LLM age.

[5] https://en.wikipedia.org/wiki/Fundamental_theorem_of_arithmetic

At the bachelor level. The main goal is to teach young students to have a good grasp, both intuitive and formal, of logic, semantics, proofs, modelling, software tools like SAT-solvers, first-order ATPs (Automated Theorem Provers), and LP. One can start with propositional logic, introducing natural deduction (ND) as the proof formalism. From our experience, students are rather interested in the step-by-step proof construction process of ND. It demystifies and desacralizes the notion of proof. Moreover, students can see that proofs are computational objects that can be checked or pretty-printed (*e.g.*, in HTML and TEX format as is done with LPTP), *i.e.*, they can be algorithmically processed.

Then one moves to FOL to conclude with LP and LPTP. In our opinion, it would be beneficial to first focus on declarative LP, possibly using CLP, before Prolog programming. From our experience, declarative LP is much easier to understand by both students and semantic software analyzers. Through various termination criteria (*e.g.*, bounded non-determinism [8]), the operational aspects could be delegated to the LP system. The art of Prolog (as, for example, advocated in [9]) remains of course in writing declarative LP code running efficiently on a standard Prolog engine.

At this point, one might consider introducing a basic methodology for LLM usage, aiming at automatically writing simple code, tests and first LPTP proofs for elementary properties. In our opinion, it is crucial to closely monitor the process. A Prolog engine runs tests. LPTP checks the generated proofs. Students review generated code, tests, and properties thereby exercising their critical thinking. Can we simplify the code? Do the tests cover the intended usage? Do the LPTP properties make sense?

At the master level. For graduate students, after reviewing the notions studied at the bachelor level, the curriculum could aim at introducing more advanced tools like abstract interpretation for invariant generation, using an LLM for help in formulating more complex LPTP statements and proofs, and ATPs for cheaper resolution proofs of LPTP statements. As demonstrated in [10, 7], these tools[6] speed up the certification process of logic programs. Students could work in small groups on certified LP projects to be evaluated by an oral assessment.

In this way, the emphasis is mainly on the semantic level, while the syntactic effort, which can be overwhelming for beginners (*e.g.*, writing long formal proofs), is largely computer-assisted.

Some remarks:

- LPTP is also an IDE (Integrated Development Environment) where one can write and proof-check ND proofs not only about properties of logic programs but also in propositional logic and FOL using the *same syntax*.
- Originally integrated into Emacs, LPTP is now available as an autonomous web page thanks to the efforts of the Ciao-Prolog team. Using a recent web browser like Chrome, pointing the navigator to the right URL[7] loads an *autonomous browser-based LP IDE*, including Ciao-Prolog, its associated abstract interpreter Ciao-PP, and LPTP. Source code can freely mix Prolog code, Ciao-PP assertions and LPTP properties and proofs.
- Once an LLM has been introduced, it will be difficult to go back. Hence we think that it is important to be able to solve elementary exercises manually in order to improve the understanding of the involved concepts. We need to find attractive ways implementing this requirement.
- Beyond their technical capabilities, LLMs raise economic, social, and ecological issues that students should be made aware of during their studies.
- In parallel, a similar methodology could be taught for programming languages from different programming paradigms, *e.g.*, Dafny [11].

[6] SMT-solvers are also very useful provers, yet to be integrated in the LP/LPTP/LLM ecosystem.
[7] https://ciao-lang.org/playground/lptp.html

8. Related work

The recent arrival and rapid adoption of LLMs and their growing aptitude at writing and/or dealing with code has triggered different AI assisted programming practices or methodologies. We can distinguish the following, even though in practice they tend to overlap and hybrid approaches exist [12, 13, 14].

Among these methodologies, *vibe-coding* [15, 3] is presumably the most widespread and most easily accessible technique, for novices as well as experienced professionals. Vibe-coding is mostly understood as the process in which the user provides the LLM with a prompt representing an informal specification of a problem written in natural language, the LLM produces a solution in code, and the user tests and ideally accepts (possibly after multiple iterations) the given solution [16, 3]. While the burden of validation lies with the user, preliminary reviews in the literature argue that testing is often skipped or even delegated to the AI [16]. Consequently, benchmarks show that code generated by vibe-coding often exhibits functional correctness issues and sometimes raises serious security concerns [16, 17].

A natural approach to alleviate some of the issues presented by vibe-coding is to augment the process with formal specification and verification (see, a.o., [12, 14]). In what is sometimes called *vericoding* [4], the user provides a *formal* specification of the problem upfront, thereby limiting accessibility of the technique to a more expert audience. The LLM then generates a verifiable implementation of the specification, i.e. it accompanies the generated code with a machine-checkable proof of its correctness. While the use of formal verification can in principle eliminate the functional correctness issues sometimes encountered in vibe-coding [4, 12], correctness (and other) guarantees are obviously highly dependent on the quality and completeness of the specification. Available benchmarks [4, 18] show results depending on the formal language used. In these benchmarks, using Dafny seems to perform better than Verus and Lean [4, 18], but the effective success rates vary widely and, as the authors note, are subject to the rapid progress of the LLM models themselves. While these results are encouraging, other results are less optimistic. For example, in [19], the authors report very low success rates when evaluating a more demanding benchmark in which both the specification and the implementation are required to be generated and verified via Lean.

While in vibe-coding and vericoding a human user orchestrates the code (and, in the case of vericoding, the proof) generation, *agentic coding* [20, 21] goes further in the sense that the human user formulates a higher-level goal, and delegates to an autonomous AI agent a large part of the software production process (coding, developing and running tests, fixing bugs, and up to submitting pull requests) without human intervention. Verification is mostly empirical. While some speedups are reported, these appear highly context and task dependent [22, 23]. Moreover, the use of agents also appears to induce persistent quality risks. A recent analysis [24] reports that roughly half of agentic pull requests (for code passing automated tests) was rejected by the maintainers of a repository.

9. Conclusion

In this paper, we report on an experiment in which the first third of the well-known P-99 set of Prolog exercises is solved using LPTP in combination with an LLM. We used Claude (Anthropic) in Code mode with the Opus 4.6 model. By solving an exercise, we mean reading the natural language specification, providing the Prolog code, constructing and running a test file, and stating and proving the usual properties of the Prolog code (types, groundness, termination, existence, unicity). For functional properties, we interacted with the LLM and checked each output. The time needed to solve an exercise varies from 15 minutes (P01) to several hours (P35). An interesting observation is that the generated code and proofs of P35 define an LP (logic programming) view of the *prime factorization theorem* of arithmetic.

The *vibe-coding* part of the experiment—reading the specification, and writing the Prolog code and tests—was handled easily by the LLM. We first checked these outputs to ensure that we were working with the correct code, free from impure constructs such as the cut and from the use of Prolog's native numbers.

The *vericoding* part of the experiment—stating and proving reliability guarantees of the generated code—was clearly more challenging for the LLM; however, it was generally able to handle it, except for functional properties. One should keep in mind that the properties proved by the LLM have been the subject of decades of research work in program verification and abstract interpretation [25], in particular in the LP setting [26]. We have now many theoretical results and some implementations (*e.g.*, [27, 28]) which compute more precise results than those inferred by the LLM. We have already begun investigating the use of automated theorem provers to verify properties of Prolog programs expressed in LPTP [29], as well as the automatic generation of LPTP proofs for groundness properties [10].

Nonetheless, the ease and rapidity with which the LLM was able to grasp the (largely unknown) LPTP formalism are impressive. We even observed that, when faced with a difficult proof, Claude examined the Prolog source code of the proof checker to gain a deeper "understanding" of the proof-checking process and fix the problematic proof. Also combining an LLM with a proof-checker avoids the "hallucinations" that the LLM may produce and allows it to fix its errors. Thus, the LPTP/LLM combination appears to be a valuable tool for the LP developer. This applies both to constructing LPTP proofs of invariants inferred by abstract interpretation—when no purely algorithmic method is available—and to proving functional properties of Prolog code, where the corresponding abstract domain is typically not implemented as it depends on the specific problem (see Section 6).

Interesting problems to investigate are the *scalability* of this approach to certified logic programming and how to teach it to students, starting for instance from initial ideas sketched in this paper.

Declaration on Generative AI

During the preparation of this work, the author(s) used Claude (Anthropic) in order to: generate the Prolog code, the test files and the LPTP proofs for the P-99 exercises, as described throughout this paper. After using this tool, the author(s) reviewed and edited the content, ran the tests, proof-checked every proof with LPTP, and take(s) full responsibility for the publication's content.

References

[1] J. F. Morales, S. Abreu, D. Ferreiro, M. V. Hermenegildo, Teaching Prolog with Active Logic Documents, in: D. S. Warren, V. Dahl, T. Eiter, M. V. Hermenegildo, R. A. Kowalski, F. Rossi (Eds.), Prolog: The Next 50 Years, Lecture Notes in Computer Science, Springer, 2023, pp. 171–183. URL: https://doi.org/10.1007/978-3-031-35254-6_14. doi:10.1007/978-3-031-35254-6_14.

[2] R. F. Stärk, The theoretical foundations of LPTP (a logic program theorem prover), Journal of Logic Programming 36 (1998) 241–269.

[3] A. Sarkar, A. Drosos, Vibe coding: programming through conversation with artificial intelligence, in: Proceedings of the 36th Annual Conference of the Psychology of Programming Interest Group (PPIG 2025), 2025. URL: https://arxiv.org/abs/2506.23253.

[4] S. Bursuc, T. Ehrenborg, S. Lin, L. Astefanoaei, I. E. Chiosa, J. Kukovec, A. Singh, O. Butterley, A. Bizid, Q. Dougherty, M. Zhao, M. Tan, M. Tegmark, A benchmark for vericoding: formally verified program synthesis, CoRR abs/2509.22908 (2025). URL: https://doi.org/10.48550/arXiv.2509.22908. doi:10.48550/ARXIV.2509.22908. arXiv:2509.22908.

[5] R. F. Stärk, First-order theories for pure Prolog programs with negation, Arch. Math. Log. 34 (1995) 113–144. URL: https://doi.org/10.1007/BF01270391. doi:10.1007/BF01270391.

[6] R. F. Stärk, Total correctness of logic programs: A formal approach, in: R. Dyckhoff, H. Herre, P. Schroeder-Heister (Eds.), ELP'96, volume 1050 of *LNCS*, Springer, 1996, pp. 237–254. URL: https://doi.org/10.1007/3-540-60983-0_17. doi:10.1007/3-540-60983-0_17.

[7] F. Mesnard, É. Payet, W. Vanhoof, Case study: proving $\sqrt{2}$ irrational with LPTP and an LLM, in: Technical Communications, ICLP 2026, 2026.

[8] D. Pedreschi, S. Ruggieri, Bounded nondeterminism of logic programs, Ann. Math. Artif. Intell.

42 (2004) 313–343. URL: https://doi.org/10.1023/B:AMAI.0000038175.53999.3a. doi:10.1023/B:AMAI.0000038175.53999.3A.

[9] L. Sterling, E. Shapiro, The Art of Prolog - Advanced Programming Techniques, 2nd Ed, MIT Press, 1994.

[10] T. Marianne, F. Mesnard, É. Payet, Automated certification of logic program groundness analysis, in: S. Escobar, L. Titolo (Eds.), Logic-Based Program Synthesis and Transformation - 35th International Symposium, LOPSTR 2025, Rende, Italy, September 9-10, 2025, Proceedings, Lecture Notes in Computer Science, Springer, 2025, pp. 113–122. URL: https://doi.org/10.1007/978-3-032-04848-6_7. doi:10.1007/978-3-032-04848-6_7.

[11] D. Banerjee, O. Bouissou, S. Zetzsche, Dafnypro: LLM-assisted automated verification for Dafny programs, CoRR abs/2601.05385 (2026). URL: https://doi.org/10.48550/arXiv.2601.05385. doi:10.48550/ARXIV.2601.05385. arXiv:2601.05385, pOPL'26.

[12] J. Mitchell, Y. Shaaban, Position: Vibe coding needs vibe reasoning: Improving vibe coding with formal verification, in: Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages, LMPL '25, Association for Computing Machinery, New York, NY, USA, 2025, p. 84–90. URL: https://doi.org/10.1145/3759425.3763390. doi:10.1145/3759425.3763390.

[13] R. Sapkota, K. I. Roumeliotis, M. Karkee, Vibe coding vs. agentic coding: Fundamentals and practical implications of agentic AI, 2025. URL: https://arxiv.org/abs/2505.19443. arXiv:2505.19443.

[14] S. Wang, Vibecontract: The missing quality assurance piece in vibe coding, 2026. URL: https://arxiv.org/abs/2603.15691. arXiv:2603.15691.

[15] C. Meske, T. Hermanns, E. V. der Weiden, K.-U. Loser, T. Berger, Vibe coding as a reconfiguration of intent mediation in software development: Definition, implications, and research agenda, IEEE Access 13 (2025) 213242–213259. URL: https://api.semanticscholar.org/CorpusID:280338092.

[16] A. Fawzy, A. Tahir, K. Blincoe, Vibe coding in practice: Motivations, challenges, and a future outlook – a grey literature review, in: Proceedings of the 48th International Conference on Software Engineering (ICSE 2026), Software Engineering in Practice (SEIP), 2025. URL: https://arxiv.org/abs/2510.00328.

[17] S. Zhao, D. Wang, K. Zhang, J. Luo, Z. Li, L. Li, Is vibe coding safe? benchmarking vulnerability of agent-generated code in real-world tasks, 2026. URL: https://arxiv.org/abs/2512.03262. arXiv:2512.03262.

[18] H. Zhao, Z. Yang, J. Li, D. He, Z. Li, C. Jin, V. V. Veeravalli, A. Gupta, S. Arora, Algoveri: An aligned benchmark for verified code generation on classical algorithms, 2026. URL: https://arxiv.org/abs/2602.09464. arXiv:2602.09464.

[19] A. Thakur, J. Lee, G. Tsoukalas, M. Sistla, M. Zhao, S. Zetzsche, G. Durrett, Y. Yue, S. Chaudhuri, CLEVER: A curated benchmark for formally verified code generation, in: 2nd AI for Math Workshop @ ICML 2025, 2025. URL: https://openreview.net/forum?id=pqNFDA2TFm.

[20] A. E. Hassan, H. Li, D. Lin, B. Adams, T.-H. Chen, Y. Kashiwa, D. Qiu, Agentic software engineering: Foundational pillars and a research roadmap, CoRR abs/2509.06216 (2025). URL: https://doi.org/10.48550/arXiv.2509.06216.

[21] H. Wang, J. Gong, H. Zhang, J. Xu, Z. Wang, Ai agentic programming: A survey of techniques, challenges, and opportunities, 2025. URL: https://arxiv.org/abs/2508.11126. arXiv:2508.11126.

[22] S. Agarwal, H. He, B. Vasilescu, AI IDEs or autonomous agents? measuring the impact of coding agents on software development, 2026. URL: https://arxiv.org/abs/2601.13597. arXiv:2601.13597.

[23] J. Becker, N. Rush, E. Barnes, D. Rein, Measuring the impact of early-2025 AI on experienced open-source developer productivity, 2025. URL: https://arxiv.org/abs/2507.09089. arXiv:2507.09089.

[24] P. Whitfill, C. Wu, J. Becker, N. Rush, Many SWE-bench-passing PRs would not be merged into main, https://metr.org/notes/2026-03-10-many-swe-bench-passing-prs-would-not-be-merged-into-main/, 2026.

[25] P. Cousot, R. Cousot, Abstract interpretation: a unifed lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. of the 4th Symp. on Principles of

Programming Languages, ACM, 1977, pp. 238–252.
[26] P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, Journal of Logic Programming 13 (1992) 103–179.
[27] M. V. Hermenegildo, G. Puebla, F. Bueno, P. López-García, Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor), Sci. Comput. Program. 58 (2005) 115–140. URL: https://doi.org/10.1016/j.scico.2005.02.006. doi:10.1016/J.SCICO.2005.02.006.
[28] É. Payet, F. Mesnard, Nontermination inference of logic programs, ACM Transactions on Programming Languages and Systems 28 (2006) 256–289. doi:10.1145/1119479.1119481.
[29] F. Mesnard, T. Marianne, É. Payet, Automated theorem proving for Prolog verification, Electronic Proceedings in Theoretical Computer Science 439 (2026) 469–481. URL: http://dx.doi.org/10.4204/EPTCS.439.32. doi:10.4204/eptcs.439.32.
[30] J. W. Lloyd, Foundations of Logic Programming, Springer-Verlag, 1987.
[31] ISO/IEC 13211-1, Information Technology – Programming Languages – Prolog – Part 1: General Core, 1995.
[32] K. L. Clark, Negation as failure, in: H. Gallaire, J. Minker (Eds.), Logic and Databases, Plenum Press, New York, 1978, pp. 293–322.

A. A quick summary of LPTP

Let P be a pure logic program where negative literals may appear in the body of clauses (also called *normal program* in [30]). For sake of conciseness, we do not consider built-in predicates (see [2] for a full treatment) other than the equality `=/2`. We start with $\mathcal{L}$, the first-order language associated to P. The *goals* of $\mathcal{L}$ are:

$$G, H ::= \texttt{true} \mid \texttt{fail} \mid s = t \mid A \mid \texttt{\textbackslash+}\ G \mid (G, H) \mid (G; H) \mid \texttt{some}\ x\ G$$

where s and t are two terms, x is a variable and A is an atomic goal. The goals of $\mathcal{L}$ have the operational semantics specified by ISO-Prolog [31] assuming the occurs check.

$\hat{\mathcal{L}}$ is the specification language of LPTP. For each user-defined predicate symbol R, $\hat{\mathcal{L}}$ does not include R, but instead it contains three predicate symbols R^s, R^f, R^t of the same arity as R, which respectively express success, failure and termination of R. $\hat{\mathcal{L}}$ also contains a unary constraint for groundness gr, expressing that its argument is ground. The *formulas* of $\hat{\mathcal{L}}$ are:

$$\phi, \psi ::= \top \mid \bot \mid s = t \mid R(\vec{t}) \mid \neg\phi \mid \phi \wedge \psi \mid \phi \vee \psi \mid \phi \rightarrow \psi \mid \forall x \phi \mid \exists x \phi$$

where $\vec{t}$ is a sequence of n terms and R denotes a n-ary predicate symbol of $\hat{\mathcal{L}}$. The semantics of $\hat{\mathcal{L}}$ is the first-order predicate calculus of classical logic.

For any of the user-defined logic procedures R in a logic program P, $D_R^P(\vec{x})$ denotes its Clark's *if-and-only-if* completed definition, c.f. [32, 30].

For defining the declarative semantics of logic programs, Stärk uses three syntactic operators **S**, **F** and **T** which map goals of $\mathcal{L}$ into $\hat{\mathcal{L}}$-formulas. Intuitively, $\mathbf{S}G$ means G succeeds (any breadth-first evaluation of G succeeds), $\mathbf{F}G$ means G fails and $\mathbf{T}G$ means G terminates (the ISO-Prolog evaluation produces a finite number of answers then stops). Moreover, termination implies a safe use of negation (ground at runtime). The definition of the operators follows:

$\mathbf{S}R(\vec{t}) := R^s(\vec{t})$	$\mathbf{S}\ \texttt{true} := \top$	$\mathbf{S}\ \texttt{fail} := \bot$	$\mathbf{S}(s = t) := (s = t)$
$\mathbf{S}\texttt{\textbackslash+}G := \mathbf{F}G$	$\mathbf{S}(G, H) := \mathbf{S}G \wedge \mathbf{S}H$	$\mathbf{S}(G; H) := \mathbf{S}G \vee \mathbf{S}H$	$\mathbf{S}(\texttt{some}\ x\ G) := \exists x \mathbf{S}G$
$\mathbf{F}R(\vec{t}) := R^f(\vec{t})$	$\mathbf{F}\ \texttt{true} := \bot$	$\mathbf{F}\ \texttt{fail} := \top$	$\mathbf{F}(s = t) := \neg(s = t)$
$\mathbf{F}\texttt{\textbackslash+}G := \mathbf{S}G$	$\mathbf{F}(G, H) := \mathbf{F}G \vee \mathbf{F}H$	$\mathbf{F}(G; H) := \mathbf{F}G \wedge \mathbf{F}H$	$\mathbf{F}(\texttt{some}\ x\ G) := \forall x \mathbf{F}G$

$\mathbf{T}R(\vec{t}) := R^t(\vec{t})$ $\quad$ $\mathbf{T}\,\mathtt{true} := \top$
$\mathbf{T}\,\mathtt{fail} := \top$ $\quad$ $\mathbf{T}(s = t) := \top$
$\mathbf{T}\backslash\!+G := \mathbf{T}G \wedge gr(G)$ $\quad$ $\mathbf{T}(G, H) := \mathbf{T}G \wedge (\mathbf{F}G \vee \mathbf{T}H)$
$\mathbf{T}(G; H) := \mathbf{T}G \wedge \mathbf{T}H$ $\quad$ $\mathbf{T}(\mathtt{some}\ x\ G) := \forall x \mathbf{T}G$

With LPTP, we prove properties of a logic program P w.r.t. its *inductive extension* IND(P) which includes Clark's completion [32] and induction along the definition of the predicates. Stärk shows that the inductive extension is always consistent and proves various correctness and completeness results w.r.t. the operational semantics of Prolog [2]. The first-order theory IND(P) (cf. [2], pp. 253–254) is defined by nine axiom schemas which we describe now. We omit the fixed point axioms for builtins.

The first two axioms specify some properties of the trees built from the function symbols extracted from the program under consideration. The third axiom forbids infinite trees.

1. $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \rightarrow x_i = y_i$ [if f is n-ary and $1 \leq i \leq n$]
2. $f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m)$ [if $n \neq m$ or $f \not\equiv g$]
3. $t \neq x$ [if x occurs in t and $t \not\equiv x$]

LPTP deals with *non-ground* terms, as any ISO-Prolog processor does. LPTP offers a predefined predicate $gr/1$ that we can consider as a constraint. This relation is useful for instance for dealing with negation as failure as LPTP only allows negation by failure for *ground* goals (see the definition $\mathbf{T}\backslash\!+G$).

4. gr(c) [if c is a constant]
5. $\mathrm{gr}(x_1) \wedge \ldots \wedge \mathrm{gr}(x_m) \leftrightarrow \mathrm{gr}(f(x_1, \ldots, x_m))$ [f is m-ary]

LPTP considers each user-defined predicate through three points of view: failure, success and termination. These three viewpoints are linked with the following axioms.

6. $\neg(R^s(\vec{x}) \wedge R^f(\vec{x}))$ [if R is a user-defined predicate]
7. $R^t(\vec{x}) \rightarrow (R^s(\vec{x}) \vee R^f(\vec{x}))$ [if R is a user-defined predicate]

Axiom 6 says that for any tuple of (possibly non-ground) terms, we cannot have at the same time success and failure of R. Axiom 7 states that given termination, we have success or failure. Altogether, it means that for any tuple of terms $\vec{x}$, assuming termination, either $R(\vec{x})$ succeeds or (exclusively) $R(\vec{x})$ fails.

8. $R^s(\vec{x}) \leftrightarrow \mathbf{S}D_R^P(\vec{x}),\ R^f(\vec{x}) \leftrightarrow \mathbf{F}D_R^P(\vec{x}),\ R^t(\vec{x}) \leftrightarrow \mathbf{T}D_R^P(\vec{x})$

We recall that $D_R^P(\vec{x})$ denotes the definition of the completion [32] of the user-defined procedure $R(\vec{x})$ in the logic program P. For instance, the first equivalence $R^s(\vec{x}) \leftrightarrow \mathbf{S}D_R^P(\vec{x})$ defines $R^s(\vec{x})$.

Finally, for any property of the form $\forall\vec{x}[R^s(\vec{x}) \rightarrow \phi(\vec{x})]$, where $R(\vec{x})$ is a user-defined procedure and $\phi(\vec{x})$ an $\hat{\mathcal{L}}$ -formula, we have an induction schema. The interactive prover LPTP is able to *dynamically* generate an induction axiom on demand while the user interacts with it. Let us examine a simple case. It is exactly what happens using LPTP, which slightly generalizes [2]. By *directly recursive user-defined predicate* in the box below, we forbid mutual recursive definitions. Of course, LPTP is able to handle mutually recursive properties, see [5] for some examples. Let R be a directly recursive user-defined predicate and let $\phi(\vec{x})$ be an $\hat{\mathcal{L}}$-formula such that the length of $\vec{x}$ is equal to the arity of R. Let $sub(\phi(\vec{x})/R)$ be the formula to be proven $\forall\vec{x}(R^s(\vec{x}) \rightarrow \phi(\vec{x}))$. Let $closed(\phi(\vec{x})/R)$ be the formula obtained from $\forall\vec{x}(\mathbf{S}D_R^P(\vec{x}) \rightarrow R^s(\vec{x}))$ by replacing

- $R^s(\vec{x})$ by $\phi(\vec{x})$ on the right of $\rightarrow$,
- all occurrences of $R(\vec{t})$ appearing on the left of $\rightarrow$ by $\phi(\vec{t}) \wedge R(\vec{t})$.

Then the induction axiom is the following formula:

9. $closed(\phi(\vec{x})/R) \rightarrow sub(\phi(\vec{x})/R)$

Together, these nine axiom schemas define the inductive extension IND(P) on which every LPTP proof relies. This concludes our brief overview of the formalism. We refer the reader to the papers of Stärk [5, 6, 2] for a complete presentation of LPTP, including the treatment of some built-in predicates.

#	Property	Statement
quot/3 — quotient		
1	Termination	$\forall d_0, n, q.\ \texttt{nat}(d_0) \wedge \texttt{nat}(n) \Rightarrow \textbf{terminates}\ \texttt{quot}(s(d_0), n, q)$
2	Types	$\forall d_0, n, q.\ \texttt{nat}(d_0) \wedge \texttt{nat}(n) \wedge \texttt{quot}(s(d_0), n, q) \Rightarrow \texttt{nat}(q)$
3	Strict bound	$\forall d_1, n, q.\ \texttt{nat}(d_1) \wedge \texttt{nat}(n) \wedge \texttt{quot}(s(s(d_1)), n, q) \Rightarrow q < n$
4	Correctness	$\forall d_0, n, q.\ \texttt{nat}(d_0) \wedge \texttt{nat}(n) \wedge \texttt{quot}(s(d_0), n, q) \Rightarrow s(d_0) \times q = n$
5	Positive	$\forall d, n, q.\ \texttt{quot}(d, n, q) \Rightarrow \exists q_0.\ q = s(q_0)$
6	Success	$\forall d, n.\ \texttt{nat}(d) \wedge \texttt{nat}(n) \wedge \texttt{divides}(d, n) \Rightarrow \exists q.\ \texttt{quot}(d, n, q)$
7	Uniqueness	$\forall d_0, n, q_1, q_2.\ \texttt{nat}(d_0) \wedge \texttt{nat}(n) \wedge \texttt{quot}(s(d_0), n, q_1) \wedge \texttt{quot}(s(d_0), n, q_2) \Rightarrow q_1 = q_2$
smallest_factor/4		
8	Termination	$\forall k, d_0, n, f.\ \texttt{nat}(k) \wedge \texttt{nat}(d_0) \wedge \texttt{nat}(n) \Rightarrow \textbf{terminates}\ \texttt{smallest\_factor}(n, s(d_0), k, f)$
9	Types	$\forall k, d_0, n, f.\ \texttt{nat}(k) \wedge \texttt{nat}(d_0) \wedge \texttt{nat}(n) \wedge \texttt{smallest\_factor}(n, s(d_0), k, f) \Rightarrow \texttt{nat}(f)$
10	Lower bound	$\forall k, d_0, n, f.\ \texttt{nat}(k) \wedge \texttt{smallest\_factor}(n, s(d_0), k, f) \Rightarrow \exists f_0.\ f = s(f_0)$
11	Lower bound 2	$\forall k, d_0, n, f.\ \texttt{nat}(k) \wedge \texttt{smallest\_factor}(n, s(s(d_0)), k, f) \Rightarrow \exists f_1.\ f = s(s(f_1))$
12	Divides	$\forall k, d_0, n, f.\ \texttt{nat}(k) \wedge \texttt{smallest\_factor}(n, s(d_0), k, f) \Rightarrow \texttt{divides}(f, n)$
13	Completeness	$\forall k, d_0, n, last.\ \texttt{nat}(k) \wedge \texttt{nat}(d_0) \wedge \texttt{nat}(n) \wedge \texttt{plus}(d_0, k, last) \wedge \texttt{divides}(s(last), n) \Rightarrow \exists f.\ \texttt{smallest\_factor}(n, s(d_0), k, f)$
14	Leq factor	$\forall k, d_0, n, f, g.\ \texttt{nat}(k) \wedge \texttt{nat}(d_0) \wedge \texttt{nat}(n) \wedge \texttt{smallest\_factor}(n, s(d_0), k, f) \wedge \texttt{divides}(g, n) \wedge s(d_0) \leq g \wedge \texttt{nat}(g) \Rightarrow f \leq g$
15	Uniqueness	$\forall k, d_0, n, f_1, f_2.\ \texttt{nat}(k) \wedge \texttt{nat}(d_0) \wedge \texttt{nat}(n) \wedge \texttt{smallest\_factor}(n, s(d_0), k, f_1) \wedge \texttt{smallest\_factor}(n, s(d_0), k, f_2) \Rightarrow f_1 = f_2$
16	Is prime	$\forall k, n, f.\ \texttt{nat}(k) \wedge \texttt{nat}(n) \wedge \texttt{smallest\_factor}(n, s(s(0)), k, f) \Rightarrow \texttt{is\_prime}(f)$
divides/2 — auxiliary		
17	Self	$\forall d.\ \texttt{divides}(d, d)$
18	Sum	$\forall d, n, b, c.\ \texttt{divides}(d, n) \wedge \texttt{divides}(d, b) \wedge \texttt{nat}(n) \wedge \texttt{nat}(b) \wedge \texttt{nat}(d) \wedge \texttt{plus}(n, b, c) \Rightarrow \texttt{divides}(d, c)$
19	Times factor	$\forall f_0, g, q.\ \texttt{nat}(f_0) \wedge \texttt{nat}(q) \wedge \texttt{nat}(g) \wedge \texttt{divides}(g, q) \Rightarrow \texttt{divides}(g, s(f_0) \times q)$
20	Transitive	$\forall a, b, c.\ \texttt{nat}(a) \wedge \texttt{nat}(b) \wedge \texttt{nat}(c) \wedge \texttt{divides}(a, b) \wedge \texttt{divides}(b, c) \Rightarrow \texttt{divides}(a, c)$
prime_factors/2		
21	Termination	$\forall n, l.\ \texttt{nat}(n) \Rightarrow \textbf{terminates}\ \texttt{prime\_factors}(s(n), l)$
22	Types	$\forall n, l.\ \texttt{nat}(n) \wedge \texttt{prime\_factors}(n, l) \Rightarrow \texttt{list}(l)$
23	Nat members	$\forall n, l.\ \texttt{nat}(n) \wedge \texttt{prime\_factors}(n, l) \Rightarrow (\forall z.\ \texttt{member}(z, l) \Rightarrow \texttt{nat}(z))$
24	Nat list	$\forall n, l.\ \texttt{nat}(n) \wedge \texttt{prime\_factors}(n, l) \Rightarrow \texttt{nat\_list}(l)$
25	Head info	$\forall n, h, t.\ \texttt{nat}(n) \wedge \texttt{prime\_factors}(n, [h\|t]) \Rightarrow \texttt{divides}(h, n) \wedge \texttt{nat}(h) \wedge (\exists f_1.\ h = s(s(f_1)))$
26	Product	$\forall n, l.\ \texttt{nat}(n) \wedge \texttt{prime\_factors}(n, l) \Rightarrow \texttt{product}(l, n)$
27	Ordered	$\forall n, l.\ \texttt{nat}(n) \wedge \texttt{prime\_factors}(n, l) \Rightarrow \texttt{ordered}(l)$
28	All prime	$\forall n, l, z.\ \texttt{nat}(n) \wedge \texttt{prime\_factors}(n, l) \wedge \texttt{member}(z, l) \Rightarrow \texttt{is\_prime}(z)$
29	Existence	$\forall n.\ \texttt{nat}(n) \Rightarrow \exists l.\ \texttt{prime\_factors}(s(n), l)$
30	Uniqueness	$\forall n, l_1, l_2.\ \texttt{nat}(n) \wedge \texttt{prime\_factors}(n, l_1) \wedge \texttt{prime\_factors}(n, l_2) \Rightarrow l_1 = l_2$

Table 4: Properties proved for `prime_factors/2`