Babel-formal: Translation of Proofs between Lean and Rocq

Théo Stoskopf
Inria, CNRS, ENS de Lyon,
Université Claude Bernard Lyon 1,
LIP, UMR 5668, Lyon, France
theo.stoskopf@inria.fr

Cyril Cohen
Inria, CNRS, ENS de Lyon,
Université Claude Bernard Lyon 1,
LIP, UMR 5668, Lyon, France
cyril.cohen@inria.fr

Nicolas Tabareau
Nantes Université, École Centrale Nantes,
CNRS, Inria,
LS2N, UMR 6004, Nantes, France
nicolas.tabareau@inria.fr

Abstract

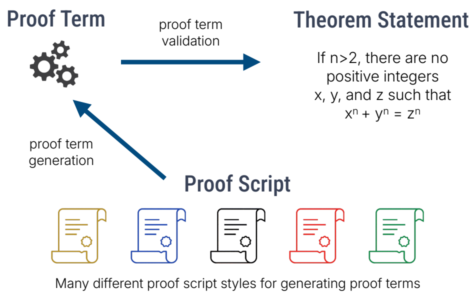
In this work, we investigate using proof terms (the low-level representation of formal proofs) as a pivot language for translating proof scripts between proof assistants and across tactic sets. Unlike direct proof translation, this approach does not require an aligned training corpus; it only needs aligned context at inference time so that both systems elaborate comparable terms. We compare two strategies: (1) direct script-to-script translation with an off-the-shelf LLM (GPT-5), and (2) proof term translation, where an LLM turns a proof term into a proof script in the target language. We build a small benchmark of aligned sources (14 files, 117 lemmas) across Lean and Rocq, and train models to map Lean and Rocq proof terms back to their native proof scripts. Our experiments show that proof term translation works for cross-assistant translation and for translation between tactic sets. It is complementary to using a SoTA off-the-shelf LLM for direct proof script translation (combining both performs best), scales easily in terms of training data, and handles tactic-set translation better (*e.g.,* vanilla Rocq → SSReflect).

1 Introduction

Proof assistants such as Rocq and Lean accumulate over time large libraries of formal mathematics (for example, MathComp [8] for Rocq and Mathlib [2] for Lean), yet proofs are not reusable across systems, even in a strictly aligned context. We study two translation settings. The first is *cross-assistant* translation between Lean and Rocq, which is useful to import the many results that exist in one prover but not in the other. The second is *cross-tactic* translation inside Rocq, from the standard Rocq tactic language — called here *vanilla Rocq* — to *SSReflect* [6], which is a compact, structured tactic language widely used with MathComp (see an example in Appendix A.5), which is useful to make existing proofs more robust to changes.

Both Rocq and Lean provide a way to pretty print *proof terms* given by the proof assistant (see Fig. 1) in a way that omits details while keeping the structure of a proof. We thus explore using *pretty-printed proof terms*, as a pivot language. This removes the need for an aligned training corpus of tactic scripts. At inference time, we only require strictly aligned statements — *i.e.,* same theorem and premises — so that both systems elaborate comparable terms. The approach is related to decompilation in programming languages, where models map assembly to higher-level C programs [13].

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.



Given a formal statement of a theorem, proving it amounts to constructing a proof term—a precise object that the prover can type-check for correctness. However, these proof terms are typically extremely verbose, capturing every low-level detail, and are impractical to write directly by hand.
To address this, users interact with interactive theorem provers through proof scripts—which are essentially meta-programs designed to generate these proof terms. Over time, this meta-programming layer has evolved in various ways, leading to a diverse ecosystem of proof script languages; often referred to as tactic languages.

Figure 1: How interactive theorem provers operate

We compare two strategies:

1. Direct script translation. An LLM is prompted to convert a proof script directly from source tactics to target tactics. We apply this in both settings (Lean $\leftrightarrow$ Rocq and vanilla Rocq $\rightarrow$ SSReflect), using GPT-5 [10] as the baseline.

2. Proof term translation. We first extract the proof term, then train models to predict scripts from proof terms (Lean term to Lean script, Rocq term to Rocq script). At inference, we extract the term for the source proof and use the model to produce a script in the target assistant or tactic set (see Fig. 2). For an example of Lean $\rightarrow$ Rocq inference see Appendix A.6, and Appendix A.5 for vanilla Rocq $\rightarrow$ SSReflect.

To support this comparison, we build a benchmark of 14 files with 117 lemmas that are strictly aligned between Lean and Rocq. We also mention, but do not pursue, direct low-level term translation between Lean and Rocq [12]. The main drawback of that direction is that the output is not tied to a maintainable proof script.

In a nutshell, proof term translation can be leveraged for translation between proof assistants (*e.g.,* Lean and Rocq) or between different sets of tactics (for example, vanilla Rocq and SSReflect). It is also easy to scale in terms of data, since no aligned training data are needed. Direct tactic translation and proof term translation are complementary, and combining them works best. Code, data, and models: github.com/LLM4Rocq/babel-formal.

2 Methodology

We describe how we build the training data, the inference pipeline, and benchmarks. To build the training data we follow four steps [9]: (1) select a diverse subset of proofs, (2) extract proof terms in Rocq and Lean, (3) generate synthetic backward reasoning traces inferring proof script from proof term , and (4) train models that predict a tactic script from a proof term. Full prompt templates and training details are in Appendix A.2 and Appendix A.1.

Proof Subset Selection. Following S1 [9], we create in each case a training set of 1000 entries. For translation between Lean and Rocq, we select a subset of 500 diverse proofs in C-CoRN [3] and 500 diverse proofs in Mathlib. For translation from vanilla Rocq to SSReflect, we select a subset of 1000 entries in MathComp. In both cases, we filter by length, then choose a diverse subset (statements and proofs) using `BM25`[11].

Proof Term Extraction. For Rocq, we use `coqpyt` [1] to extract, for each proof, its proof term together with the premises and notation declarations. For Lean 4, we use `leanclient` [5] to extract the proof term and the context.

Backward Reasoning Trace Generation. Given each proof term and the corresponding target script, we generate backward reasoning traces with Gemini 2.5 Pro [4]. Backward reasoning trace generation refers here to prompting the LLM with the proof term and the target proof script, and asking it to generate a reasoning process that works backwards from the script. The resulting training dataset follows the structure described in Fig. 3a.

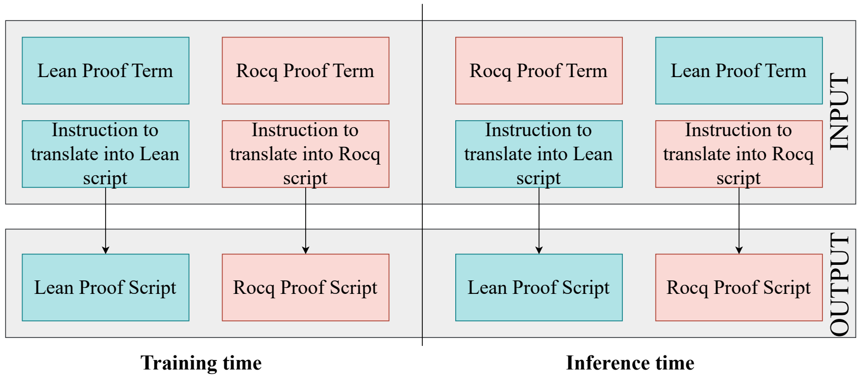
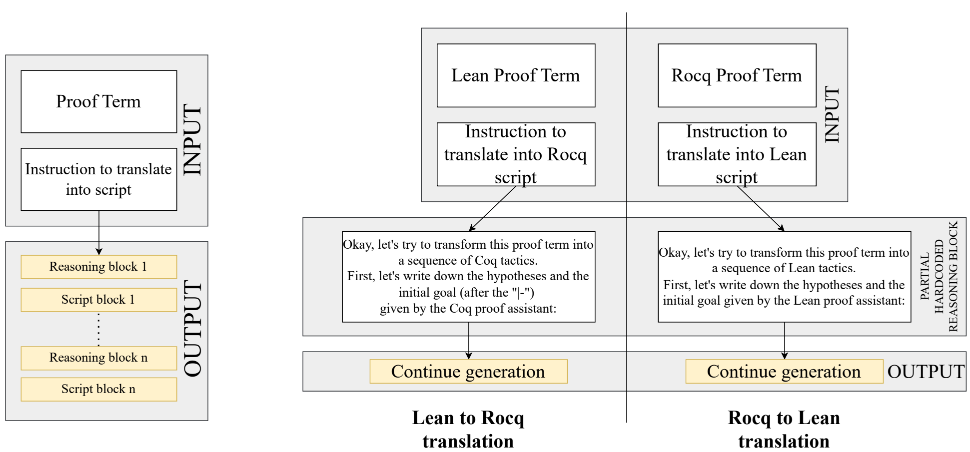


Figure 2: Training and Inference pipeline

Benchmarks. We evaluate on two benchmarks:

Cross-assistant (Lean $\leftrightarrow$ Rocq). We created 14 files with the same statements in Rocq and Lean, for a total of 117 aligned lemmas. Preparing files took about 20 hours of manual effort in total. Each lemma is proved natively in both systems, so the proof scripts differ but are type-correct in aligned contexts. From these, we extracted comparable proof terms in both systems. An example of such a source file is in Appendix A.4.

Cross-tactic (vanilla Rocq $\rightarrow$ SSReflect). We collect 500 Rocq proofs from the C-CoRN library [3], which is a large mathematical library developed in vanilla Rocq (in contrast to MathComp, which is built on SSReflect). We use these proofs to assess translation from vanilla Rocq tactics to SSReflect. To ensure diversity, we pick subsets maximizing document distance (via BM25). An example of entry is in Appendix A.5.



(a) Input/Output structure (b) Translation strategy

Figure 3: Training & Inference strategy

Inference Pipeline. At inference, we leverage the source proof term and use the appropriate strategy to obtain a script in the target assistant or tactic set (Fig. 3). The first block in Fig. 3 is initialized with the goal state and fixed instructions; the full prompt appears in Appendix A.2.

3 Experimental Setup

Models and Training. We fine-tune `Qwen2.5-Coder-32b-Instruct` [7] to translate a proof term into a tactic script. We train two models in two different scenarios: (i) **Babel-translate:** Lean term

$\rightarrow$ Lean script and Rocq term $\rightarrow$ Rocq script. (ii) **Babel-ssreflect:** Rocq term $\rightarrow$ SSReflect script. These models are used at inference to produce target scripts from source terms (Fig. 2).

Evaluation. We run verification through Rocq tooling (coq-lsp and Pétanque) [14, 15] and Lean tooling (leanclient) [5]. We use two generation strategies:

Non-interactive (Babel). For *Babel-translate* and *Babel-ssreflect*, we sample k candidates per input and report pass@k (success if any candidate verifies). No error feedback is provided.

Interactive repair (GPT-5 and Qwen2.5-32b-Coder-Instruct). For direct tactic translation baselines, the model receives the proof assistant error on failure and may attempt to repair/continue the proof for up to 4 rounds. We report pass@1 with feedback, denoted "@1x(4 feedback rounds)" for GPT-5. For Qwen2.5-32b-Coder-Instruct we additionally sample a small pool per round (*e.g.,*, 16 candidates), denoted "@16x(4 feedback rounds)", and count success if any candidate in any round verifies (see Appendix A.3 for the prompt template)

Tasks. We evaluate three directions: (i) Lean $\rightarrow$ Rocq, (ii) Rocq $\rightarrow$ Lean (cross-assistant), and (iii) vanilla Rocq $\rightarrow$ SSReflect (cross-tactic), using the benchmarks introduced in Section 2. For (i) and (ii) we apply *Babel-translate* (proof term $\rightarrow$ native script) and compare with direct GPT-5 translation. For (iii) we apply *Babel-ssreflect* (proof term $\rightarrow$ SSReflect script), again comparing with direct GPT-5 translation.

4 Results and Analysis

Setup	**Lean4 → Rocq**	**Rocq → Lean4**	**Rocq → SSReflect**
GPT-5 (@1x(4 feedbacks))	**82.9**%	**67.5**%	16.6%
Babel-translate (@128)	68.3%	40.2%	
Babel-ssreflect (@128)			**33.2**%
Babel-ssreflect w/o reas. (@128)			21.8%
Babel + GPT-5 combined	89.7%	83.7%	34%
Qwen-2.5-Coder-32b-Instruct (@16x(4 feedbacks))	6.8%	19.7%	0%

Table 1: Benchmark results for different translation setups across three tasks.

In the proof script translation baseline, GPT-5 achieves 82.9% success in the direction Lean $\rightarrow$ Rocq, and 67.5% in Rocq $\rightarrow$ Lean. However, on the cross-tactic task vanilla Rocq $\rightarrow$ SSReflect, it manages only 16.6% success. In contrast, the cross-tactic translation model attains 33.2% success on Rocq $\rightarrow$ SSReflect, demonstrating a clear gap in style transfer. In cross-assistant tasks, term translation model reaches 68.3% on Lean $\rightarrow$ Rocq and 40.2% on Rocq $\rightarrow$ Lean.

Ablation study. Removing the reasoning component in *Babel-ssreflect* reduces Rocq $\rightarrow$ SSReflect from 33.2% to 21.8%. For Lean $\leftrightarrow$ Rocq, the model trained without reasoning produced mixed scripts that were not usable, so we do not report scores.

Limitation. Babel models do not receive interactive feedback during proof generation. They see the initial goal, the context, and the proof term only, then must produce a complete script. Interactive repair, as used by GPT-5, may helps in some cases.

5 Conclusion and Future Work

We introduced proof term translation as a way to translate proofs across assistants and tactic styles. Unlike direct tactic translation, this approach does not need parallel scripts for training. On our benchmarks it brings clear gains for style transfer and remains competitive for cross-assistant translation. It is complementary to direct translation, and combining the two gives the best results.

Future work. We see three directions: (i) **Scale.** Train on larger corpora and more domains, since the approach does not require aligned scripts. (ii) **Feedback.** Add type checker feedback during generation. A simple loop that feeds proof assistant errors into the reasoning blocks could guide local repairs and improve robustness. (iii) **Source-to-source translation (Lean $\leftrightarrow$ Rocq).** Use off-the-shelf LLMs for source-to-source translation, and bootstrap proof script translation to check whether the source translation is correct.

6 Acknowledgments

This project was funded by the Inria "Défi" LLM4Code. This work was granted access to the HPC resources of IDRIS under the allocation 2025-AD011016066 made by GENCI.

References

[1] Pedro Carrott, Nuno Saavedra, Kyle Thompson, Sorin Lerner, João F. Ferreira, and Emily First. CoqPyt: Proof navigation in python in the era of LLMs. *arXiv preprint arXiv:2405.04282*, 2024.

[2] Mathlib Community. The lean mathematical library. *arXiv preprint arXiv:1912.09375*, 2019.

[3] Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive coq repository at nijmegen. In *Mathematical Knowledge Management (MKM 2004)*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2004.

[4] Google DeepMind. Gemini 2.5 Pro: Advanced multimodal model. `https://ai.googleblog.com/2024/07/gemini-25-our-newest-gemini-model-with.html`, 2024.

[5] Oliver Dressler. Leanclient: Python client to interact with the lean4 language server, 1 2025.

[6] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. Ssreflect: Small scale reflection in Coq. Coq documentation.

[7] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, Yunlong Feng, Xingzhang Ren, Xuancheng Ren, Jingren Zhou, and Junyang Lin. Qwen2.5-Coder technical report. 2024.

[8] Assia Mahboubi and Enrico Tassi. Mathematical components. `https://math-comp.github.io`, 2020.

[9] Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Fei-Fei Li, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. S1: Simple test-time scaling. *arXiv preprint arXiv:2405.00002*, 2025.

[10] OpenAI. Introducing GPT-5. `https://openai.com/blog/introducing-gpt-5`, 2025.

[11] S. E. Robertson and K. Sparck Jones. Relevance weighting of search terms. *Journal of the American Society for Information Science*, 27(3):129–146, May 1976.

[12] The rocq-lean-import Development Team. Coq is a lean typechecker. `https://github.com/rocq-community/rocq-lean-import`, 2020.

[13] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2406.00004*, 2024.

[14] The Coq LSP Development Team. Coq-LSP: Language server protocol implementation for the Coq proof assistant. `https://github.com/ejgallego/coq-lsp`, 2024.

[15] Laetitia Teodorescu, Guillaume Baudart, Emilio Jesús Gallego Arias, and Marc Lelarge. Nlir: Natural language intermediate representation for mechanized theorem proving. In *MathAI@NeurIPS*, 2024.

A Appendix

A.1 Training Details

We fine-tune a pretrained and instruction-tuned `Qwen2.5-32b-Coder-Instruct`[7] for reasoning, following the setup of S1 simple test-time scaling [9]. We train for five epochs with a global batch size of 16, which yields 315 gradient steps. We use bfloat16 precision and an AdamW optimizer with $\beta_1 = 0.9$, $\beta_2 = 0.95$, and weight decay 1×10^{-4}. The learning rate is 1×10^{-5}, warmed up linearly for the first 5% (16 steps) and then cosine-decayed to zero over the remaining 299 steps. We compute loss only on the reasoning traces and the final solutions, and we choose a sequence length large enough to avoid truncation. On GENCI hardware (8×4 NVIDIA H100), this run completes in about one hour. At inference, we sample up to 128 candidates.

A.2 Prompt Babel-translate

For Lean → Rocq, we instruct the model to translate a Lean proof term and its local context into a Rocq tactic script. We prepend the following instruction:

```
You are given a proof term:

{term}

Your task is to derive a sequence of tactics that corresponds to this term.

When you work through the problem, write down your reasoning in detail inside
<think> ... </think> tags. This reasoning should reflect your natural thought
process as you explore the structure of the term and figure out what tactics
to apply. You should consider different possible approaches, reflect on why
some might or might not work, and gradually converge on a tactic choice.

After each reasoning block, provide the next (group of) tactic(s) enclosed in:

\box{{
  <tactic>
}}

Some dependencies that could be helpful:

{dependencies}
```

We hard-code the beginning of the first thinking block:

```
<think> Okay, let's try to transform this proof term into a sequence of coq
tactics.
First let's write down the hypotheses, and the initial goal
(after the "|-" symbol) given by the coq proof assistant:
{initial_goal}.
```

A.3 Prompt GPT-5 direct translation

For Lean → Rocq, we instruct the model to translate a Lean proof script into a Rocq proof script. We give the following instruction:

```
You are a precise translator from Lean 4 proofs to Rocq (Coq).

## INPUT
<ROCQ_STATEMENT>
{rocq_statement}
</ROCQ_STATEMENT>
```

```
<LEAN_STATEMENT>
{lean_statement}
</LEAN_STATEMENT>

<LEAN_PROOF>
{lean_proof}
</LEAN_PROOF>

<DEPENDENCIES>
{dependencies}
</DEPENDENCIES>

## RULES
1) Produce two blocks in this exact order and format:
   a) A thought process.
   b) The final Rocq proof script delimited by the markers below.
2) Use only lemmas/results listed in <DEPENDENCIES>. Do not rely on
   external lemmas beyond these. Standard Rocq/Coq tactics are allowed.
3) Do not invent new lemmas.
4) Do not use "Admitted.". End with "Qed.".
5) Output must contain only the two blocks below, nothing else.

## OUTPUT FORMAT (STRICT)
<BEGIN_RATIONALE>
[Thought process.]
</END_RATIONALE>

<BEGIN_ROCQ_PROOF>
[Place only the Rocq/Coq script here. Inside "Proof." ... "Qed.",
do not rewrite the Rocq statement.]
</END_ROCQ_PROOF>

Now produce the output for the given input.
```

In case of an issue, we prepend the following feedback to the next prompt:

```
You already tried this proof:
{proof}
but it fails with this error:
{error_message}.
```

A.4 Benchmark example

We show one Lean/Rocq pair from the aligned set. The Rocq entry is:

```
Class AbsField := {
  R0   : Type;
  zero : R0;
  one  : R0;
  add  : R0 -> R0 -> R0;
  mul  : R0 -> R0 -> R0;
  opp  : R0 -> R0;
  abs  : R0 -> R0;
  leR  : R0 -> R0 -> Prop;
  ltR  : R0 -> R0 -> Prop;


  N0       : Type;
  Nle      : N0 -> N0 -> Prop;
  Nmax     : N0 -> N0 -> N0;
```

```
  Nle_max_l : forall x y, Nle x (Nmax x y);
  Nle_max_r : forall x y, Nle y (Nmax x y);

  add_comm  : forall x y, add x y = add y x;
  add_assoc : forall x y z, add (add x y) z = add x (add y z);
  add_zero  : forall x, add x zero = x;
  add_opp   : forall x, add x (opp x) = zero;
  mul_comm  : forall x y, mul x y = mul y x;
  mul_assoc : forall x y z, mul (mul x y) z = mul x (mul y z);
  mul_one   : forall x, mul x one = x;

  le_refl   : forall x, leR x x;
  le_trans  : forall x y z, leR x y -> leR y z -> leR x z;
  add_le_add : forall a b c d, leR a b -> leR c d -> leR (add a c)\\
  (add b d);

  abs_nonneg : forall x, leR zero (abs x);
  abs_triangle : forall x y, leR (abs (add x y)) (add (abs x) (abs y));
  abs_opp : forall x, abs (opp x) = abs x;
  abs_sub_symm : forall x y, abs (add x (opp y)) = abs (add y (opp x));
  sub_decomp : forall x y z, add x (opp z) = add (add x (opp y)) \\
  (add y (opp z));
  sub_eq_zero : forall x y, add x (opp y) = zero -> x = y;


  eq_of_forall_eps2 : forall x, (forall eps, ltR zero eps -> leR \\
  (abs x) (add eps eps)) -> x = zero
}.

Section Limits.
  Context {A : AbsField}.

  Definition sub (x y : R0) : R0 := add x (opp y).

  Definition limit (u : N0 -> R0) (l : R0) : Prop :=
    forall eps, ltR zero eps -> exists N, forall n, Nle N n -> leR\\
    (abs (sub (u n) l)) eps.

  Lemma abs_sub_triangle (x y z : R0) :
    leR (abs (sub x z)) (add (abs (sub x y)) (abs (sub y z))).
  Proof.
    unfold sub.
    rewrite (sub_decomp x y z).
    apply abs_triangle.
  Qed.

  Theorem limit_unique (u : N0 -> R0) (l m : R0) :
    limit u l -> limit u m -> l = m.
  Proof.
    intros Hl Hm.
    assert (Hbound: forall eps, ltR zero eps -> leR (abs (sub l m))\\
    (add eps eps)).
    { intros eps Heps.
      destruct (Hl eps Heps) as [N1 HN1].
      destruct (Hm eps Heps) as [N2 HN2].
      set (N := Nmax N1 N2).
      assert (H1u: leR (abs (sub (u N) l)) eps).
      { apply HN1. apply Nle_max_l. }
      assert (H1: leR (abs (sub l (u N))) eps).
```

```
    { unfold sub in H1u. rewrite abs_sub_symm in H1u. fold sub\\
    in H1u. exact H1u. }
    assert (H2: leR (abs (sub (u N) m)) eps).
    { apply HN2. apply Nle_max_r. }
    assert (Htri: leR (abs (sub l m)) (add (abs (sub l (u N)))\\
    (abs (sub (u N) m)))) by apply abs_sub_triangle.
    eapply le_trans; [exact Htri|].
    apply add_le_add; [exact H1|exact H2].
  }
  assert (Hz: sub l m = zero).
  { apply eq_of_forall_eps2. exact Hbound. }
  unfold sub in Hz.
  now apply sub_eq_zero in Hz.
Qed.

End Limits.
```

The Lean counterpart is:

```
class AbsField (R : Type) where
  zero  : R
  one   : R
  add   : R → R → R
  mul   : R → R → R
  opp   : R → R
  abs   : R → R
  leR    : R → R → Prop
  ltR    : R → R → Prop


  NatAlt     : Type
  NatAltle   : NatAlt → NatAlt → Prop
  NatAltMax  : NatAlt → NatAlt → NatAlt
  le_max_left  : forall x y, NatAltle x (NatAltMax x y)
  le_max_right : forall x y, NatAltle y (NatAltMax x y)

  add_comm  : forall x y, add x y = add y x
  add_assoc : forall x y z, add (add x y) z = add x (add y z)
  add_zero  : forall x, add x zero = x
  add_opp   : forall x, add x (opp x) = zero
  opp_add   : forall x y, opp (add x y) = add (opp x) (opp y)
  mul_comm  : forall x y, mul x y = mul y x
  mul_assoc : forall x y z, mul (mul x y) z = mul x (mul y z)
  mul_one   : forall x, mul x one = x

  le_refl   : forall x, leR x x
  le_trans  : forall x y z, leR x y → leR y z → leR x z
  add_le_add : forall a b c d, leR a b → leR c d → leR (add a c) \\
  (add b d)

  abs_nonneg : forall x, leR zero (abs x)
  abs_triangle : forall x y, leR (abs (add x y)) (add (abs x) (abs y))
  abs_opp : forall x, abs (opp x) = abs x
  abs_sub_symm : forall x y, abs (add x (opp y)) = abs (add y (opp x))


  sub_decomp : forall x y z, add x (opp z) = add (add x (opp y)) \\
  (add y (opp z))
  sub_eq_zero : forall x y, add x (opp y) = zero → x = y
```

```
  eq_of_forall_eps2 : forall x, (forall eps, ltR zero eps → leR (abs x) \\
  (add eps eps)) → x = zero

namespace Limits

variable {R : Type} [AR : AbsField R]

def sub (x y : R) : R := AbsField.add x (AbsField.opp y)

def limit (u : (AbsField.NatAlt (R := R)) → R) (l : R) : Prop :=
  forall eps : R, AbsField.ltR AbsField.zero eps →
    exists N : AbsField.NatAlt (R := R),
      forall n : AbsField.NatAlt (R := R),
        AbsField.NatAltle (R := R) N n →
          AbsField.leR (AbsField.abs (sub (u n) l)) eps

theorem abs_sub_triangle (x y z : R) :
  AbsField.leR (AbsField.abs (sub x z)) (AbsField.add (AbsField.abs \\
  (sub x y)) (AbsField.abs (sub y z))) := by

  have : sub x z = AbsField.add (sub x y) (sub y z) := sub_decomp x y z
  simpa [this] using AbsField.abs_triangle (sub x y) (sub y z)

theorem limit_unique (u : (AbsField.NatAlt (R := R)) → R) (l m : R) :
  limit u l → limit u m → l = m :=
by
  intro Hl Hm

  have Hbound' : forall eps, AR.ltR AR.zero eps → AR.leR (AR.abs (sub l m)) \\
  (AR.add eps eps) := by
    intro eps Heps
    rcases Hl eps Heps with ⟨N1, HN1⟩
    rcases Hm eps Heps with ⟨N2, HN2⟩
    let N := AbsField.NatAltMax (R := R) N1 N2
    have H1 : AbsField.leR (AbsField.abs (sub (u N) l)) eps := by
      apply HN1
      exact AbsField.le_max_left (R := R) _ _
    have H2 : AbsField.leR (AbsField.abs (sub (u N) m)) eps := by
      apply HN2
      exact AbsField.le_max_right (R := R) _ _

    have Htri : AbsField.leR (AbsField.abs (sub l m)) (AbsField.add \\
    (AbsField.abs (sub l (u N))) (AbsField.abs (sub (u N) m))) := by
      simpa using abs_sub_triangle l (u N) m


    have H1' : AbsField.leR (AbsField.abs (sub l (u N))) eps := by

      simpa [sub, AbsField.abs_sub_symm (u N) l] using H1
    exact AbsField.le_trans _ _ _ Htri (AbsField.add_le_add _ _ _ _ H1' H2)

  have Hz : sub l m = AbsField.zero := AbsField.eq_of_forall_eps2 (sub l m) Hbound'

  exact AbsField.sub_eq_zero l m (by simpa [sub] using Hz)
end Limits
```

A.5 Translation to SSReflect

We illustrate the Rocq → SSReflect translation on a lemma extracted from CoRN. Babel takes as input the proof term and produces a short SSReflect script that closes the goal under the same context.

Lemma (CoRN).

```
Lemma CR_nonZero_as_Cauchy_IR_nonZero_1 :
  forall (x:CR), (CRapartT x 0)%CR ->
  Dom (f_rcpcl' _) (CRasCauchy_IR x).
```

Proof term.

```
CR_nonZero_as_Cauchy_IR_nonZero_1 =
fun (x : CR) (x_ : (x >< ' 0%Q)%CR) =>
  ap_wdr_unfolded Cauchy_IR (CRasCauchy_IR x) (CRasCauchy_IR (' 0%Q)%CR) [0]
    (CR_ap_as_Cauchy_IR_ap_1 x (' 0%Q)%CR x_)
    (CR_inject_Q_as_Cauchy_IR_inject_Q 0)
  : forall x : CR,
    (x >< ' 0%Q)%CR ->
    Dom (f_rcpcl' Cauchy_IR) (CRasCauchy_IR x)

Arguments CR_nonZero_as_Cauchy_IR_nonZero_1 x _
```

Original Rocq script.

```
intros x x_.
change ((CRasCauchy_IR x)[#][0]).
stepr (CRasCauchy_IR 0%CR).
apply CR_ap_as_Cauchy_IR_ap_1.
assumption.
eapply CR_inject_Q_as_Cauchy_IR_inject_Q.
Defined.
```

Babel (SSReflect) script from the term.

```
move=> x_; rewrite /CRapartT; apply: CR_ap_as_Cauchy_IR_ap_1 => /=.
```

A.6 Translation from Lean to Rocq

We illustrate the Lean → Rocq translation on a lemma whose Lean proof uses the `simp` tactic. Babel takes as input the proof term and the goal (it does not see the Lean proof), and reconstructs the explicit calls to lemmas such as `map_smul` and `smul_zeroV` that `simp` applies implicitly.

Lemma (Lean).

```
theorem ker_smul {L : LMap (R:=R) V W} {a : R} {x : V} :
  ker L x → ker L (VSpace.smul (R:=R) a x) :=
by
  intro hx
  unfold ker at *
  simp [hx]
```

Proof term.

```
ker_smul =
fun (VS2 : VSpace) (L : Lin) (a : R0) (x : V0) (hx : ker L x) =>
  eq_ind_r (fun v : V0 => v = zeroV)
    (eq_ind_r (fun v : V0 => (a •V v) = zeroV)
      (eq_ind_r (fun v : V0 => v = zeroV) eq_refl (smul_zeroV a)) hx)
```

```
    (map_smul L a x)
  : forall {VS2 : VSpace} (L : Lin) (a : R0) (x : V0),
    ker L x -> ker L (a •V x)
```

Original Lean script.

```
intro hx
unfold ker at *
simp [hx]
```

Babel (Rocq) reconstruction. The beginning of the first reasoning block (initial hypotheses and goal) is hardcoded by us to provide context; it is *not* generated by the model.

```
Okay, let's try to transform this proof term into a sequence of Coq tactics.
First, let's write down the hypotheses and the initial goal (after the "|-")
given by the Coq proof assistant:

F  : Field
VS1, VS2  : VSpace
L  : Lin
a  : R0
x  : V0
|- ker L x -> ker L (a •V x)
```

```
The proof term starts with fun {R} ... {x} hx => .... This is a lambda
abstraction that binds the variable hx. In a Coq proof, this corresponds
to introducing an assumption. The type of this assumption is the premise
of the implication in the goal, which is ker L x. So, the first logical
step is to introduce this assumption into our proof context. The tactic
for this is intros. The lambda term gives it the name hx, so I'll use
that. This will change our goal to the conclusion of the implication,
which is ker L (a •V x). </think>
\box{intros hx.}

<think> Now, my proof state is:
F, VS1, VS2, L, a, x : ...
hx : ker L x
|- ker L (a •V x)

The goal is ker L (a •V x). This is a definition. To make progress, I
need to understand what I'm trying to prove. ker is the kernel of a
linear map, which means it's the set of vectors that the map sends to zero.
So, ker L (a •V x) should mean L (a •V x) = 0. To make the proof term
easier to work with, I should unfold this definition. The tactic for
unfolding definitions in the goal is unfold. Let's apply that to ker in the
goal. </think>
\box{unfold ker in |- *.}

<think> The goal has now unfolded to L (a •V x) = Lin.VSpace.zeroV R. The
lambda term contains L.map_smul a x. This looks like a lemma about linear
maps and scalar multiplication. It probably states that L (a •V x) =
a •V (L x). This seems like a very useful fact, as it lets me distribute the
linear map L over the scalar multiplication •V. My current goal's left-hand
side is L (a •V x), which is the perfect form to apply this lemma. The tactic
rewrite is used to apply an equality to the goal. Since map_smul is a lemma
of the form LHS = RHS, rewrite map_smul will replace LHS with RHS. I expect
this to change my goal to a •V (L x) = Lin.VSpace.zeroV R. This looks like a
promising simplification. </think>
```

```
\box{rewrite map_smul.}

<think> My goal is now a •V (L x) = Lin.VSpace.zeroV R. Let's look at my
hypotheses. I have hx : ker L x. This is still using the ker definition.
It would be helpful to have this hypothesis in its unfolded form as well,
L x = Lin.VSpace.zeroV R. This will make it easier to use hx to rewrite parts
of my goal. I'll use the unfold tactic again, this time on the hypothesis hx.
</think>
\box{unfold ker in hx.}

<think> Now my context contains hx : L x = Lin.VSpace.zeroV R, and my goal is
a •V (L x) = Lin.VSpace.zeroV R. This is great. The term L x in my goal appears
exactly as the left-hand side of the equality in my hypothesis hx. This is a
perfect opportunity to use the rewrite tactic. By doing rewrite hx, I can
replace L x in the goal with Lin.VSpace.zeroV R. I expect this to transform
my goal into a •V (Lin.VSpace.zeroV R) = Lin.VSpace.zeroV R. </think>
\box{rewrite hx.}

<think> The goal is now a •V (Lin.VSpace.zeroV R) = Lin.VSpace.zeroV R. Looking
at the lambda term, I see a reference to Lin.VSpace.smul_zero a. This must be
the lemma for a •V 0 = 0. My goal's left-hand side, a •V (Lin.VSpace.zeroV R),
is precisely the form required by this lemma. Once again, rewrite is the tactic
to use. Applying rewrite smul_zeroV should simplify the left-hand side of my
goal to Lin.VSpace.zeroV R, making the
goal Lin.VSpace.zeroV R = Lin.VSpace.zeroV R. </think>
\box{rewrite smul_zeroV.}

<think> After the last rewrite, my goal is Lin.VSpace.zeroV R = Lin.VSpace.zeroV R.
This is an equality of a term with itself. This is a trivially true statement. There
are several tactics to solve such a goal. reflexivity is the most direct one, as it
is designed specifically for proving t = t. The lambda term structure Eq.trans ...
(eq_self ...) confirms that the proof concludes by showing both sides are the same
thing. Therefore, reflexivity is the final step to close the proof. </think>
\box{reflexivity.}
```