


Automatic Goal Clone Detection in Rocq

Ali Ghanbari 

Auburn University, AL, USA

Abstract

Proof engineering in Rocq is a labor-intensive process, and as proof developments grow in size, redundancy and maintainability become challenges. One such redundancy is *goal cloning*, i.e., proving α -equivalent goals multiple times, leading to wasted effort and bloated proof scripts. In this paper, we introduce *clone-finder*, a novel technique for detecting goal clones in Rocq proofs. By leveraging the formal notion of α -equivalence for Gallina terms, *clone-finder* systematically identifies duplicated proof goals across large Rocq codebases. We evaluate *clone-finder* on 40 real-world Rocq projects from the CoqGym dataset. Our results reveal that each project contains an average of 27.73 instances of goal clone. We observed that the clones can be categorized as either *exact goal duplication*, *generalization*, or *α -equivalent goals with different proofs*, each signifying varying levels duplicate effort. Our findings highlight significant untapped potential for proof reuse in Rocq-based formal verification projects, paving the way for future improvements in automated proof engineering.

2012 ACM Subject Classification Software and its engineering → Software maintenance tools; Software and its engineering → Formal software verification; Theory of computation → Type theory

Keywords and phrases Clone Detection, Goal, Proof, Rocq, Gallina

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.12

Supplementary Material *Software (Source code)*: <https://aub.ie/cfsrc>
archived at `swh:1:dir:3c32ef30be8294f864b7be5fd7109522fe5a2eb1`

Acknowledgements The author thanks Anonymous ECOOP 2025 Reviewers for their insightful and constructive feedback that significantly improved this paper.

1 Introduction

Rocq [20]¹ is a popular interactive theorem prover based on type theory. Rocq has proved to be a viable tool for constructing certified software, as demonstrated by projects like CompCert [27], Iris [22], ConCert [2], and Fiat Cryptography [13].

Proof engineering [33] in Rocq is a labor-intensive task [6, 15, 14]. As proof developments grow in size, redundancy and maintainability become major challenges. An important, yet relatively overlooked, issue in proof engineering research is *goal cloning* – proving of α -equivalent goals more than once. Such a duplication of effort not only results in waste of manpower, but also inflates proof scripts, increases maintenance costs, and reduces opportunities for reuse. While Rocq offers mechanisms for structuring proofs, avoiding redundancy remains a challenge. Large-scale verification projects, such as seL4 [26] and Verisoft [1], have highlighted the necessity of tools that support automated detection of redundant proof efforts [6]. While there is tool support for code clone detection [18, 40] in Isabelle/HOL [30], to the best of our knowledge, detection of redundant proof efforts in Rocq proofs is not explored yet.

The problem that we set out to solve is as follows. *Given a set of Rocq proofs, find the set of α -equivalent goals making up the proofs.* “Goal” is a Rocq terminology for proof obligation; each proof comprises sub-proofs for a set of goals that are written as Gallina [20]

¹ The Coq development community recently renamed the system to The Rocq Prover. In this paper, we use Coq and Rocq interchangeably.



© Ali Ghanbari;

licensed under Creative Commons License CC-BY 4.0

39th European Conference on Object-Oriented Programming (ECOOP 2025).

Editors: Jonathan Aldrich and Alexandra Silva; Article No. 12; pp. 12:1–12:19

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

terms. We want to automatically identify *goal clones*, *i.e.*, α -equivalent goals. Such goals, and their corresponding sub-proofs, can be factored out as independent lemmas and reused, instead of proving them from scratch each time they are observed. To solve this, we introduce *clone-finder*, a novel technique that leverages the formal definition of α -equivalence of Gallina terms to systematically identify duplicated proof goals across large Rocq codebases. This enables proof engineers to refactor their developments by extracting common subproofs as reusable lemmas, thereby reducing redundant proof effort and improving maintainability. Although the problem statement is simple, and the notion of α -equivalence of Gallina terms has a solid foundation in the well-established field of type theory, finding α -equivalent Gallina terms in practice is not trivial. This is mainly because the variables mean different things in different contexts and any comparison must first universally quantify the terms with the free variables defined in the context before performing α -equivalence check (see §3 for more details on the proposed approach).

Our prototype Python implementation of *clone-finder* interacts with Rocq through Coq-LSP [9] and its Python interface, CoqPyt [7]. We evaluate *clone-finder* on 40 real-world Rocq projects from the CoqGym dataset [42]. Our benchmark contains a diverse set of Rocq projects, representative of the real-world Rocq projects: the project are of different sizes, ranging from 53 to 29,260 lines of code, and they are from various domains – from set theory to distributed system verification to geometry (see §5.1 for more details on benchmark preparation process). We ran *clone-finder* on our benchmark of Rocq projects, while measuring the number of goal clones that it detects as well as its runtime overhead. We observed that in 20 out of 40 projects, *clone-finder* detected at least one goal clone. On average *clone-finder* finds 27.73 instances of goal clones per project. Manual review of the reported clones revealed three common patterns of goal cloning: (1) exact goal duplication with identical or near-identical proofs, *e.g.*, structurally identical proofs with different variable names; (2) α -equivalent goals with generalized proofs, where one proof subsumes the other; and (3) α -equivalent goals with entirely different proofs. Each of these instances signifies extra spent proof effort that could be saved and repurposed for other tasks by factoring out the duplicated goals as independent lemmas. These findings suggest significant untapped potential for proof reuse in Rocq-based formal verification projects.

Another observation that we make is that *clone-finder* is a light-weight system: over 94.71% of its running time is from using Coq-LSP through CoqPyt. An average 45.31 seconds of run time, for a fresh run, and support for incremental analysis, which could significantly reduce the run time, make *clone-finder* a practical tool for daily proof engineering. Mitigating the overhead of interfacing Coq-LSP is a matter of investing engineering effort so that CoqPyt does not reload common dependencies of the proof scripts within a workspace every time we load a script from that workspace. Another possibility is to implement *clone-finder* directly as a Rocq plugin using Rocq plugin library and Coq-Elpi [8]. We save these as a future extension of this work.

Beyond detecting proof redundancy, our work contributes an open-source enhancement to CoqPyt, which enables analyzing real-world Rocq projects with it. We will make both *clone-finder* and our enhancement publicly available to foster further research in automated proof engineering. To summarize, this paper makes the following contributions.

- **Technique:** a simple technique, named *clone-finder*, based on the theoretically elegant notion of α -equivalence is proposed for detecting goal clones in Rocq scripts. This technique can be used alongside lemma extraction tool-set [32, 35] to rewrite clones as independent lemmas to increase reuse and mitigate waste of proof efforts. Additionally, the support for incremental analysis would allow using *clone-finder* in a background thread in a Rocq IDE to detect clones as the proof engineer writes the proofs.

- **Evaluation:** We evaluate `clone-finder` using a set of real-world Rocq projects. We report that real-world Rocq projects are likely to contain large numbers of clones with either (1) identical; (2) generalized; or (3) entirely different pairs of proofs. These clones indicate waste of manpower in proof engineering that could potentially be repurposed for use in other aspects of system specification and verification.
- **Open-Source Contribution:** While implementing `clone-finder`, which interacts with the Python interface for Coq-LSP, named CoqPyt, we realized that CoqPyt does not allow passing physical-to-logical path mapping, which is essential in working with real-world Rocq projects. We have added this feature to the framework and will make a pull request about it. We have also made `clone-finder` publicly available [17]. This source code contains a Python-based parser for Gallina that can be used in future research projects.

The remainder of this paper is organized as follows. We provide background on Rocq and code clone detection in §2. In §3 we present the `clone-finder` approach in detail, and in §4 we discuss the implementation of our prototype. We present our experimental evaluation in §5, followed by an analysis of threats to validity in §6. We review related work in §7 and conclude the paper, while sketching future research directions, in §8.

2 Background

In this section, we briefly introduce Rocq, Gallina, Calculus of Inductive Constructions, and code clone detection. We also provide references for further reading on these topics.

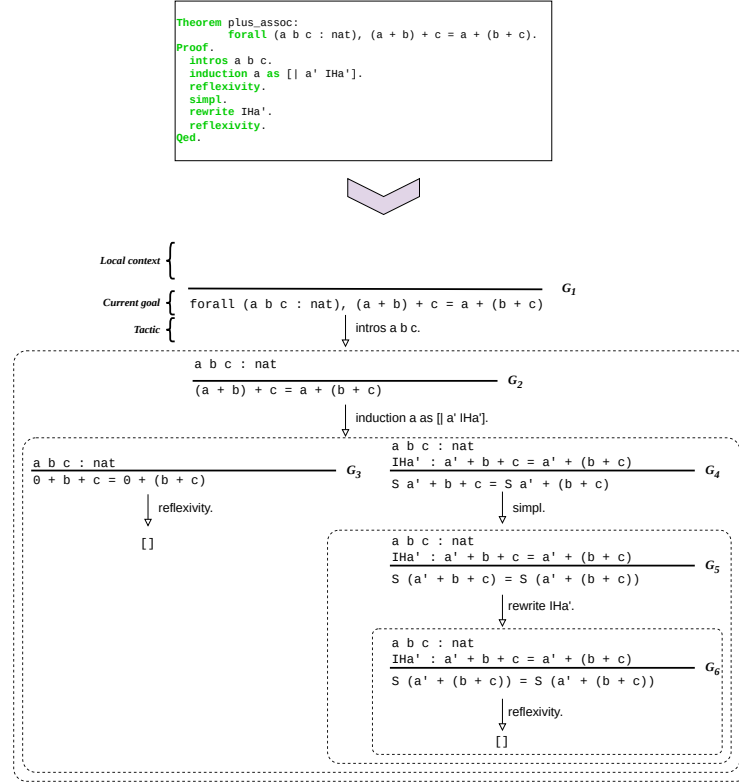
2.1 Rocq, Gallina, and Calculus of Inductive Constructions

Rocq [20], formerly known as Coq, is an interactive theorem prover. At its core, lies a specification language, named Gallina [20]. Gallina is an elegant functional programming language that was originally built upon the calculus of inductive constructions with definitions [4, 31, 39]. This calculus still constitutes a substantial subset of Gallina today. Proofs in Rocq are usually written in a tactic language, named Ltac [12, 20], which is a domain-specific language for building Gallina terms in a type-driven fashion that serve as proofs. The type of these terms are also Gallina terms, that serve as theorem statements and goals.

Rocq is a state-based interactive proof assistant. Theorem proving in Rocq starts with the theorem itself as the initial goal. The proof engineer repeatedly applies Ltac tactics to decompose the goal into a list of subgoals. The proof is complete when there are no sub-goals left. At each step, Rocq displays the local definitions as available hypotheses that can be used in theorem proving. A successful Rocq proof implicitly amounts to creating a proof tree whose root is the original theorem and whose nodes are the goals. Figure 1 shows a simple example. All goals share the same global context, but have exclusive local contexts, *e.g.*, hypotheses `a, b : nat` in Figure 1. The edges of the proof tree are annotated with tactics. The proof for a goal in the proof tree can be obtained by conducting a depth-first search [10] on the node corresponding to the goal and accumulating the tactics that are visited along the way.

2.2 Code Clone Detection

Code clone detection is an active area of research in software engineering aimed at identifying duplicated code snippets within software systems [28, 36, 29, 41]. Code duplication is widespread in software development. It is sometimes introduced deliberately to accelerate prototyping, reuse existing functionality, or meet tight deadlines, and other times it arises inadvertently. However, excessive cloning can degrade code quality, increase maintenance overhead, and propagate defects across multiple locations in the codebase [23, 24].



■ **Figure 1** A simple Rocq theorem and its proof in Ltac. The proof process starts with the theorem statement as the current goal. Each valid Ltac command transforms the state of the proof assistant, prompting it to generate zero or more new goals. This amounts to creating a proof tree, where nodes corresponds to goals and the edges are labeled with the Ltac command that generated the child nodes. A depth-first search on each node of the tree to accumulate the tactics at the edges constitute the sub-proof for the goal at each node. This figure also illustrates how Algorithm 1 constructs proof trees by executing proofs line by line, while tracking the goals that are created and eliminated as a result of running each tactic.

Code clones refer to code snippets that exhibit similarity based on a defined metric. The literature categorizes code clones into four primary types [36, 29]: (1) exact clones; (2) renamed clones; (3) near-miss clones; and (4) semantic clones.

While the code clone detection technique introduced in this paper falls under the categories of type 1 and type 2 clones (our technique finds α -equivalent goals in a project), the type of code that we study in this paper are fundamentally different from that of studied in the literature. We are finding clones in functional programs, rather than imperative programs, so the existing techniques for code clone detection [28, 29, 41] could not be used directly.

3 The clone-finder Approach

At the highest level, `clone-finder` takes, as input, the base path for a Rocq project, Rocq options, and the minimum proof size threshold, and outputs a list of clones with proofs at least as large as the provided threshold. The proof size is defined in terms of lines of Ltac code. The operations of `clone-finder` can be summarized as (1) constructing proof trees

$$\begin{aligned}
t &::= s \mid x \mid t_1 t_2 \mid \text{forall } (x : t_1), t_2 \\
&\quad \mid \text{fun } (x : t_1) \Rightarrow t_2 \mid \text{let } x : t_1 := t_2 \text{ in } t_3 \\
&\quad \mid \text{fix } y (x : t_1) : t_2 := t_3 \\
&\quad \mid \text{match } t_1 \text{ return } t_2 \text{ with } [|x_1 \dots x_n \Rightarrow t_3]^* \text{ end} \\
\\
&\text{(terms)} \quad t, t_1, t_2, t_3 \in \mathbb{T} \\
&\text{(sorts)} \quad s \in \{\text{Set}, \text{Type}\} \\
&\text{(variables)} \quad x, y, x_1, \dots, x_n \in \mathbb{V}
\end{aligned}$$

■ **Figure 2** Abstract syntax of a core subset of Gallina terms.

(described in Algorithm 1); (2) flattening, deduplicating, and generalization of the proof trees to construct the list of pairs of goals and their corresponding proofs (described in Algorithms 2 and 3); (3) finding pairs of α -equivalent generalized goals and reporting the output (described in Algorithm 4). We now describe each of these steps in more details.

The enabling tool of **clone-finder** are the notions of free variables and α -equivalence. To define these notions more precisely, we use a core subset of Gallina subsumed by the calculus of inductive constructions with definitions. Figure 2 shows this subset of Gallina. Note that while inductive type definitions are part of the syntax of the calculus of inductive constructions with definitions, we have not included them in our core language, because this is how Rocq operates: after an inductive type definition is type checked, the type name, its constructors, as well as induction principles and recursor, are added as constants into the global type environment and type definition themselves do not appear in goals. The following definitions define notions of free variables, substitution, and α -equivalence. Readers familiar with these standard notions may skip the following three definitions.

► **Definition 1 (Free Variables).** *The set of free variables of a Gallina term t , denoted $FV(t)$, is defined recursively as follows.*

- $FV(s) = \emptyset$, for all sorts s ,
- $FV(x) = \{x\}$, for all $x \in \mathbb{V}$,
- $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$ for all $t_1, t_2 \in \mathbb{T}$,
- $FV(\text{forall } (x : t_1), t_2) = FV(t_1) \cup (FV(t_2) - \{x\})$, for $x \in \mathbb{V}$ and $t_1, t_2 \in \mathbb{T}$,
- $FV(\text{fun } (x : t_1) \Rightarrow t_2) = FV(t_1) \cup (FV(t_2) - \{x\})$, for $x \in \mathbb{V}$ and $t_1, t_2 \in \mathbb{T}$,
- $FV(\text{let } x : t_1 := t_2 \text{ in } t_3) = FV(t_1) \cup FV(t_2) \cup (FV(t_3) - \{x\})$, for $x \in \mathbb{V}$ and $t_1, t_2, t_3 \in \mathbb{T}$,
- $FV(\text{fix } y (x : t_1) : t_2 := t_3) = FV(t_1) \cup FV(t_2) \cup (FV(t_3) - \{x, y\})$,
- $FV(\text{match } t_1 \text{ return } t_2 \text{ with } |C_1| \dots |C_k \text{ end}) = FV(t_1) \cup FV(t_2) \cup FV_C(C_1) \cup \dots \cup FV_C(C_k)$, for $k \geq 0$,

where s is a sort, $x, y \in \mathbb{V}$, and $t_1, t_2, t_3 \in \mathbb{T}$. The helper function FV_C is defined to be $FV_C(x_1 \dots x_n \Rightarrow t_3) = FV(t_3) - \{x_1, \dots, x_n\}$, wherein $n \geq 1$ and $x_1, \dots, x_n \in \mathbb{V}$.

All the non-free variables occurring in a Gallina term are referred to as *bound variables*. Two Gallina terms are said to be α -equivalent, if we can systematically rename bound variables in one term to make it syntactically equal to another. To define this notion precisely, we first need to define variable replacement, which is used for variable renaming in the α -equivalence check.

► **Definition 2** (Substitution). *Given a variable x and a term u , $t^{x \rightarrow u}$ denotes the substitution of x with u in t , and is defined as follows.*

- $s^{x \rightarrow u} = s$, for all sorts s ,
- $x_0^{x \rightarrow u} = u$, if $x_0 = x$,
- $x_0^{x \rightarrow u} = x_0$, for all $x_0 \in \mathbb{V}$ such that $x_0 \neq x$,
- $(t_1 \ t_2)^{x \rightarrow u} = t_1^{x \rightarrow u} \ t_2^{x \rightarrow u}$, for all $t_1, t_2 \in \mathbb{T}$,
- $(\text{forall } (x_0 : t_1), t_2)^{x \rightarrow u} = \text{forall } (x_0 : t_1^{x \rightarrow u}), t_2^{x \rightarrow u}$, for all $x_0 \in \mathbb{V}$ and $t_1, t_2 \in \mathbb{T}$ such that $x_0 \neq x$ and $x_0 \notin FV(u)$,
- $(\text{forall } (x_0 : t_1), t_2)^{x \rightarrow u} = \text{forall } (x_0 : t_1^{x \rightarrow u}), t_2$, if $x_0 \in FV(u)$,
- $(\text{fun } (x_0 : t_1) \Rightarrow t_2)^{x \rightarrow u} = \text{fun } (x_0 : t_1^{x \rightarrow u}) \Rightarrow t_2^{x \rightarrow u}$, for all $x_0 \in \mathbb{V}$ and $t_1, t_2 \in \mathbb{T}$ such that $x_0 \neq x$ and $x_0 \notin FV(u)$,
- $(\text{fun } (x_0 : t_1) \Rightarrow t_2)^{x \rightarrow u} = \text{fun } (x_0 : t_1^{x \rightarrow u}) \Rightarrow t_2$, for all $t_1, t_2 \in \mathbb{T}$ such that $x_0 \in FV(u)$,
- $(\text{let } x_0 : t_1 := t_2 \text{ in } t_3)^{x \rightarrow u} = \text{let } x_0 : t_1^{x \rightarrow u} := t_2^{x \rightarrow u} \text{ in } t_3^{x \rightarrow u}$, for all $x_0 \in \mathbb{V}$ and $t_1, t_2, t_3 \in \mathbb{T}$ such that $x_0 \neq x$ and $x_0 \notin FV(u)$,
- $(\text{let } x_0 : t_1 := t_2 \text{ in } t_3)^{x \rightarrow u} = \text{let } x_0 : t_1^{x \rightarrow u} := t_2^{x \rightarrow u} \text{ in } t_3$, for all $t_1, t_2, t_3 \in \mathbb{T}$ such that $x_0 \in FV(u)$,
- $(\text{fix } y (x_0 : t_1) : t_2 := t_3)^{x \rightarrow u} = \text{fix } y (x_0 : t_1^{x \rightarrow u}) : t_2^{x \rightarrow u} := t_3^{x \rightarrow u}$, for all $y, x_0 \in \mathbb{V}$ and $t_1, t_2, t_3 \in \mathbb{T}$ such that $y \neq x$, $x_0 \neq x$, $x_0 \notin FV(u)$, and $y \notin FV(u)$,
- $(\text{fix } y (x_0 : t_1) : t_2 := t_3)^{x \rightarrow u} = \text{fix } y (x_0 : t_1^{x \rightarrow u}) : t_2^{x \rightarrow u} := t_3$, for all $t_1, t_2, t_3 \in \mathbb{T}$ such that $x_0 \in FV(u)$ or $y \in FV(u)$,
- $(\text{match } t_1 \text{ return } t_2 \text{ with } |C_1| \dots |C_k \text{ end})^{x \rightarrow u} = \text{match } t_1^{x \rightarrow u} \text{ return } t_2^{x \rightarrow u} \text{ with } |C_1^{x \rightarrow u}| \dots |C_k^{x \rightarrow u} \text{ end}$, where $C_i^{x \rightarrow u}$ is defined as $(x_1 \dots x_n \Rightarrow t_3)^{x \rightarrow u} = (x_1 \dots x_n \Rightarrow t_3^{x \rightarrow u})$, if $x \notin \{x_1, \dots, x_n\}$ and $\{x_1, \dots, x_n\} \cap FV(u) = \emptyset$, and $(x_1 \dots x_n \Rightarrow t_3)^{x \rightarrow u} = x_1 \dots x_n \Rightarrow t_3$ if $\{x_1, \dots, x_n\} \cap FV(u) \neq \emptyset$.

We now define the notion of α -equivalence as follows.

► **Definition 3** (α -equivalence). *The following axiom and rules of inference prove judgments of the form $t_1 =_\alpha t_2$, where t_1 and t_2 are Gallina terms.*

- **(Renaming-Forall)**: $(\text{forall } (x : t_1), t_2) =_\alpha (\text{forall } (y : t'_1), t'_2)$, if $t_1 =_\alpha t'_1$, $y \notin FV(t_2)$, y is not bound in t_2 , and $t_2^{x \rightarrow y} =_\alpha t'_2$,
- **(Renaming-Fun)**: $(\text{fun } (x : t_1) \Rightarrow t_2) =_\alpha (\text{fun } (y : t'_1) \Rightarrow t'_2)$, if $t_1 =_\alpha t'_1$, $y \notin FV(t_2)$, y is not bound in t_2 , and $t_2^{x \rightarrow y} =_\alpha t'_2$,
- **(Renaming-Let)**: $(\text{let } x : t_1 := t_2 \text{ in } t_3) =_\alpha (\text{let } y : t'_1 := t'_2 \text{ in } t'_3)$, if $t_1 =_\alpha t'_1$, $t_2 =_\alpha t'_2$, $y \notin FV(t_3)$, y is not bound in t_3 , and $t_3^{x \rightarrow y} =_\alpha t'_3$,
- **(Renaming-Fix)**: $(\text{fix } y (x : t_1) : t_2 := t_3) =_\alpha \text{fix } y' (x' : t'_1) : t'_2 := t'_3$, if $t_1 =_\alpha t'_1$, $t_2 =_\alpha t'_2$, $y' \notin FV(t_3)$, y' is not bound in t_3 , $x' \notin FV(t_3)$, x' is not bound in t_3 , and $t_3^{y \rightarrow y' \ x \rightarrow x'} =_\alpha t'_3$,
- **(Renaming-Match)**: $(\text{match } t_1 \text{ return } t_2 \text{ with } |C_1| \dots |C_k \text{ end}) =_\alpha (\text{match } t'_1 \text{ return } t'_2 \text{ with } |C'_1| \dots |C'_k \text{ end})$, if $t_1 =_\alpha t'_1$, $t_2 =_\alpha t'_2$, and for all $1 \leq i \leq k$, we have $C_i \equiv_\alpha C'_i$, where \equiv_α is defined as: $(x_1 \dots x_n \Rightarrow t_3) \equiv_\alpha (x'_1 \dots x'_n \Rightarrow t'_3)$ if $x'_1 \dots x'_n \notin FV(t_3)$, variables $x'_1 \dots x'_n$ are not bound in t_3 , and $t_3^{x_1 \rightarrow x'_1 \dots x_n \rightarrow x'_n} =_\alpha t'_3$.
- **(Compatibility-1)**: If $t_1 =_\alpha t_2$, then $t_1 \ t =_\alpha t_2 \ t$, $t \ t_1 =_\alpha t \ t_2$, $(\text{forall } (x : t_1), t) =_\alpha (\text{forall } (x : t_2), t)$, and $(\text{fun } (x : t_1) \Rightarrow t) =_\alpha (\text{fun } (x : t_2) \Rightarrow t)$, for all term t ,
- **(Compatibility-2)**: If $t_1 =_\alpha t_2$, then $(\text{match } t_1 \text{ return } t \text{ with } |C_1| \dots |C_k \text{ end}) =_\alpha (\text{match } t_2 \text{ return } t \text{ with } |C_1| \dots |C_k \text{ end})$ and $(\text{match } t \text{ return } t_1 \text{ with } |C_1| \dots |C_k \text{ end}) =_\alpha (\text{match } t \text{ return } t_2 \text{ with } |C_1| \dots |C_k \text{ end})$, for all term t and match clauses C_1, \dots, C_k .

■ **Algorithm 1** Constructing proof trees from Rocq proofs.

Require: List of Rocq source files, `coq_files`, and `coqc` flags, stored in variable “flags”
Ensure: Returning a mapping from (File, Theorem) to ProofTree

```

1: tree_map ← ∅
2: for all file ∈ coq_files do
3:   pf ← connect_to_coq_lsp(file, flags)
4:   for all step ∈ pf do
5:     step.execute()
6:     if step.in_proof() then
7:       thm ← step.get_theorem()
8:       tree ← ProofTree()
9:       tree_map[(file, thm)] ← tree
10:      prev ← ∅
11:      while not step.can_close() do
12:        step.execute()
13:        curr ← ∅
14:        for all g ∈ pf.goals() do
15:          ctx ← g.get_hypotheses()
16:          goal ← g.get_type()
17:          curr ← curr ∪ {(ctx, goal)}
18:        end for
19:        if prev ≠ ∅ then
20:          gone ← {t ∈ prev | t ∉ curr}
21:          new ← {t ∈ curr | t ∉ prev}
22:          for all t ∈ gone do
23:            ctx ← t[0]
24:            goal ← t[1]
25:            children ← {c[1] | c ∈ new}
26:            tac ← step.get_tactic()
27:            tree.AddNode(goal, ctx, children, tac)
28:          end for
29:        end if
30:        prev ← curr
31:      end while
32:    end if
33:  end for
34:  close(pf)
35: end for
36: return tree_map

```

▷ Construct a proof tree for each theorem in each file
 ▷ Connect to Coq-LSP: an iterable proof file object is returned
 ▷ Sweep the current file step-by-step
 ▷ Have stepped into the proof of a theorem?
 ▷ Initialize a new proof tree
 ▷ Store the proof tree
 ▷ Track goals at the last step
 ▷ Iterate through the proof steps until no more goals remain
 ▷ Store current proof state
 ▷ Extract local and in-file context (hypotheses)
 ▷ Extract the goal type (i.e., goal statement)
 ▷ Store current goals
 ▷ Goals that disappeared
 ▷ Newly introduced goals
 ▷ Children are new goals
 ▷ Get the applied tactic
 ▷ Add node to proof tree and connect it to its children
 ▷ Update tracking state
 ▷ Close the proof file and the Coq-LSP connection
 ▷ Return the constructed proof tree map

- **(Compatibility-3):** If $t_1 =_\alpha t_2$, then $(\text{let } x : t_1 := t \text{ in } u) =_\alpha (\text{let } x : t_2 := t \text{ in } u)$, $(\text{let } x : t := t_1 \text{ in } u) =_\alpha (\text{let } x : t := t_2 \text{ in } u)$, $(\text{fix } y(x : t_1) : t \Rightarrow u) =_\alpha (\text{fix } y(x : t_2) : t \Rightarrow u)$, and $(\text{fix } y(x : t) : t_1 \Rightarrow u) =_\alpha (\text{fix } y(x : t) : t_2 \Rightarrow u)$, for all terms t and u ,
- **(Compatibility-4):** If $t_1 =_\alpha t_2$, then $(\text{forall } (x : t), t_1) =_\alpha (\text{forall } (z : t), t_2)$, $(\text{fun } (x : t) \Rightarrow t_1) =_\alpha (\text{fun } (z : t) \Rightarrow t_2)$, for all term t and variable z ,
- **(Compatibility-5):** If $t_1 =_\alpha t_2$, then $(\text{let } x : t := u \text{ in } t_1) =_\alpha (\text{let } z : t := u \text{ in } t_2)$, $(\text{fix } y(x : t) : u \Rightarrow t_1) =_\alpha (\text{fix } z(x : t) : u \Rightarrow t_2)$, and $(\text{fix } y(x : t) : u \Rightarrow t_1) =_\alpha (\text{fix } y(z : t) : u \Rightarrow t_2)$, and $(\text{match } t \text{ return } u \text{ with } | C_1 | \dots | (x_1 \dots x \dots x_n) \Rightarrow t_1 \dots | C_k \text{ end}) =_\alpha (\text{match } t \text{ return } u \text{ with } | C_1 | \dots | (x_1 \dots z \dots x_n) \Rightarrow t_2 \dots | C_k \text{ end})$, for all terms t and u , and variable z ,
- **(Reflexivity):** $t =_\alpha t$, for all term t ,
- **(Symmetry):** If $t_1 =_\alpha t_2$ then $t_2 =_\alpha t_1$,
- **(Transitivity):** If $t_1 =_\alpha t_2$ and $t_2 =_\alpha t_3$ then $t_1 =_\alpha t_3$.

Renaming rules are the foundations of α -equivalence. Compatibility rules have the effect of extending α -equivalence from subterms to bigger terms. Reflexivity, symmetry, and transitivity make α -equivalence an equivalence relation. If we can derive $t_1 =_\alpha t_2$, we say that the terms t_1 and t_2 are α -equivalent.

Armed with these definitions, we are now in a position to define our algorithms making up clone-finder. The first step is to extract goals and construct proof trees, which is described in Algorithm 1. This step is performed to find the proof corresponding to each goal in a Rocq project. clone-finder invokes this algorithm by first finding all the `.v` files in the target

■ **Algorithm 2** Flatten and remove redundant goals.

Require: `pt_map`, a mapping from $\langle \text{File}, \text{Theorem} \rangle$ to `ProofTree`, *i.e.*, Algorithm 1’s output
Ensure: List of goals and their generalizations, where redundant goals have been removed

```

1: g_list  $\leftarrow []$  ▷ Initialize list for storing goals
2: for all  $(f, t, pt) \in \text{pt\_map}$  do ▷ Iterate over proof tree map
3:   for all  $g \in \text{pt.GetNodes}()$  do ▷ Extract goals from proof tree
4:     ctx  $\leftarrow g.\text{GetContext}()$ 
5:     g_gen  $\leftarrow \text{Generalize}(\text{ctx}, g)$ 
6:     p  $\leftarrow g.\text{GetProof}()$ 
7:     g_list.append $((g, g\_gen, p, f, t))$ 
8:   end for
9: end for
10: red_g  $\leftarrow \{\}$  ▷ Initialize set for redundant goals
11: for  $i = 0$  to  $|\text{g\_list}| - 1$  do ▷ Iterate through goals list
12:   for  $j = 0$  to  $|\text{g\_list}| - 1$  do ▷ Compare each goal with others
13:     if  $i \neq j$  and prodBody $(\text{g\_list}[i][1], \text{g\_list}[j][1])$  then ▷ Check if one goal is part of another
14:       red_g.add $(\text{g\_list}[j][1])$ 
15:     end if
16:   end for
17: end for
18: g_list  $\leftarrow [g \in \text{g\_list} \mid g[1] \notin \text{red\_g}]$  ▷ Remove redundant goals
19: return g_list

```

Rocq project directory. This list, together with `coqc` flags are passed to the algorithm. In summary, this algorithm runs each Rocq script in the given project step by step to construct proof trees using a tree data structure. Proof tree construction is done as follows. After executing a tactic, the current goal disappears, and a set of new goals might be created, these goals are regarded as the children of the current goal in the proof tree. We can identify the edges of the tree by tracking how goals are created during the proof execution. For example, if the list of goals changes from $[A, B]$ to $[C, D, B]$, we know that node A has two children C and D . Algorithm 1 creates nodes that point to the children node with the help of a dictionary-based data structure that maps current goal to the child goals and other information about the current goals such as the tactic that transforms it to the child goals and the context right before the execution of the tactic. Figure 1 provides a simple example of a Rocq proof and its corresponding proof tree.

More precisely, Algorithm 1 constructs a mapping of Rocq scripts to their corresponding proof trees, capturing the hierarchical structure of goals and tactics used to prove them. It iterates over each Rocq source file (line 2), establishing a connection to Coq-LSP to analyze proof steps (lines 3 and 4). When a proof of a theorem is encountered (line 6), the algorithm initializes a proof tree data structure and adds it to the resulting map, mapping file name and the the current theorem name pairs to the newly created proof tree (lines 7-9). This proof tree object, gets populated as follows. The algorithm iterates through proof steps until the proof is complete (line 11). At each step, it extracts the current goals and the local, and in-file, contexts (lines 13–18), detects goals that have disappeared or emerged (lines 20–21), and records the proof structure by adding nodes to the proof tree (lines 22–29). Finally, after processing all steps, the proof file is closed (line 35), and the mapping of file names and theorem names to proof trees is returned (line 31).

The next step in `clone-finder`’s pipeline is to flatten the proof trees by turning the proof tree map generated by Algorithm 1 into a list of tuples of a goal, its generalization (*i.e.*, universal quantification of its free variables), its proof, its containing file name, and its containing theorem name. At this step, `clone-finder` also deduplicates the goals by removing the tuples whose generalization is a body of another tuple’s generalization. Algorithm 2 formalizes this process. This algorithm processes a proof tree map by flattening its structure and eliminating redundant goals. First, it initializes an empty list `g_list` (line 1) to store extracted goals and their associated information. Then, it iterates through the proof tree

■ **Algorithm 3** Generalization of a Goal.

Require: ctx: A map from variables to terms
Require: goal: The goal to be generalized
Ensure: A generalized goal with universally quantified variables

```

1: fvs ← {v ∈ FV(goal) | v ∈ ctx}           ▷ Find free variables bound in the context
2: rev_top_order ← reverse(topo_sort(ctx, fvs))   ▷ Sort dependencies in reverse topological order
3: for each v ∈ rev_top_order do
4:   goal ← “forall (“v“ : ”ctx[v]“, ”goal“)”    ▷ Prepend universal quantifiers
5: end for
6: return goal

```

map (pt_map) and extracts goals from each proof tree (lines 2–3). The method GetNodes of proof tree, returns the set of nodes, *i.e.*, the goals, in the proof tree. For each goal, it retrieves its context, generalizes it, and pairs it with its proof, file, and theorem name before appending this information to g_list (lines 4–7). The method GetProof, receives a node, *i.e.*, a goal, as input, conducts a depth-first search rooted at that node, while accumulating the tactics along each edge that it visits. The list of tactics ultimately returned by the method constitutes the proof for that goal. Another function used by the algorithm is the Generalize function, which is calculated according to Algorithm 3 and will be described shortly. Next, a set red_g is initialized (line 10) to keep track of redundant goals. The algorithm then performs a pairwise comparison of the goals (lines 11–13), marking a goal as redundant if its generalized form appears as a subterm within another goal using the function prodBody (line 13). The definition below, provides a more formal description of the prodBody function.

► **Definition 4** (prodBody Function). *Given Gallina terms t and t' , $prodBody(t, t')$ returns true if t is a product type, aka, universally quantified term, with body t'' , *i.e.*, $t = \text{forall } (x : t_0), t''$ for some term t_0 , such that either $t' = t''$ or $prodBody(t'', t')$ is true.*

Finally, the redundant goals are removed from g_list (line 18), ensuring that only distinct, non-nested goals remain.

The Generalize function transforms a given goal into a universally quantified form by identifying and universally quantifying all free variables in the goal that are present in the context (ctx). Algorithm 3 defines this function. This function first collects the relevant free variables (fvs) and determines their dependency order using topo_sort (Line 2), ensuring that variables are introduced in a correct dependency-aware sequence. The topological sorting in reverse order ensures that if variable v_1 depends on v_2 , then v_2 is quantified before v_1 , preventing type dependency issues. The function then iterates through the reverse of this topological order (Line 2) and prepends universal quantifiers (forall) to the goal in the correct order, making sure all dependent variables appear before their use. This process ensures that dependently typed terms remain well-formed while abstracting away unnecessary local dependencies.

The function topo_sort constructs a topological ordering [10] of variables in ctx that are also present in fvs. It first builds a directed acyclic graph wherein nodes are variables and edges represent dependency relationships between the variables. More precisely, assume that we have $v_1 : t_1$ and $v_2 : t_2$ in local context, where t_1 is a Gallina term such that $v_2 \in FV(t_1)$. Intuitively, this means that v_1 needs v_2 in order for it to be defined, because its type uses v_2 , so we create a directed edge from the node corresponding to v_1 to that of v_2 to denote this dependency. The function then conducts a topological sorting on this graph by conducting a depth-first search on each node. The output of topo_sort function is a list of variables that if reversed will present the variables in a valid dependency order. So, if v_1 depends on v_2 in the graph, we will universally quantify v_2 before v_1 , thereby creating a well-formed dependently typed term.

■ **Algorithm 4** Reporting clones.

Require: goals_list: List of deduplicated and generalized goals, *i.e.*, Algorithm 2’s output
Ensure: List of alpha equivalent goals

```

1: output  $\leftarrow \emptyset$ 
2: for  $i = 0$  to  $|\text{goals\_list}| - 1$  do
3:   for  $j = i + 1$  to  $|\text{goals\_list}| - 1$  do
4:     if  $\text{goals\_list}[i][1] =_{\alpha} \text{goals\_list}[j][1]$  then
5:       output.append( $\text{goals\_list}[i] \cdot \text{goals\_list}[j]$ )
6:        $\triangleright$  Concatenate goals_list[i] and goals_list[j] to generate output pairs
7:     end if
8:   end for
9: end for
10: return output

```

The last step in clone-finder’s pipeline involves finding pairs of generalized goals that are α -equivalent. This process is formalized in Algorithm 4. The algorithm simply picks pairs of tuples from the output of Algorithm 2 and checks if the compartments corresponding to the generalized goals are α -equivalent. If so, the tuples, that contain various information, such as the containing theorem name, file name, and proofs, are concatenated to form the output of clone-finder.

4 Implementation

We have implemented clone-finder in 4,969 lines of Python code. As we have given high-level explanation in §3, clone-finder interacts with Coq-LSP to open the Rocq files in a project and execute them step by step, extracting goals, as well as local context entries. In this implementation of clone-finder, for the sake of efficiency, we only consider local contexts, which could introduce unsoundness and false positives. We have implemented a Gallina parser to parse the goals and find α -equivalent goals, as outlined in §3. Currently, clone-finder offers a command-line interface through which it receives target project’s base directory, Rocq command-line options, *i.e.*, $-R/-Q$ for mapping physical paths to logical paths, and the minimum size of the proofs. clone-finder interfaces Coq-LSP through a recent Python framework, named CoqPyt [7]. We observed that CoqPy does not allow passing $-R/-Q$ options to Coq-LSP, which is essential for running real-world projects that we have studied in this paper. Therefore, we extended the API provided by the framework so that they receive $-R/-Q$ options and pass them to Coq-LSP. We shall publish this open-source contribution.

Minimum size of proofs parameter of clone-finder dictates the tool to ignore clones with proofs under a certain number of lines of Ltac code. This parameter defaults to 5, *i.e.*, only clones with proofs equal to 5 lines of Ltac commands or more will be reported. In §5, we study clone-finder’s performance with this default value. Studying clone-finder with different parameter sizes is the subject of a future extension of this work. We would also like to emphasize that clone-finder could be implemented using Coq-Elpi [8]. Our Python implementation of Gallina parser calls for disabling the notations that might have been used in the proof scripts. Specifically, in all our experiments, we invoked clone-finder by first prepending each proof script with the directive **Set Printing All** to disable the notations. This is essential for our parser to work. Despite this limitation, we opted for using Python and directly parsing Gallina expressions, as we believe that this approach would make this infrastructure available for a wider research community and foster research in automated theorem proving.

5 Experiments

In this section, we report our experimental results for running `clone-finder` on a number of real-world Rocq projects. All our experiments are conducted on a Dell workstation with a 48-core Intel Xeon Gold CPU @ 2.3 GHz and 512 GB of RAM, running Ubuntu 22.04.5 LTS.

5.1 Benchmark Projects

Our benchmark suite consists of real-world Rocq projects from the CoqGym dataset [42], which is a widely used dataset in the literature [42, 38, 37, 16, 15, 14]. CoqGym comprises 135 Rocq projects obtained from GitHub. We observed that the projects were compatible with 5 different versions of Rocq. Older projects were compatible with Coq version 8.5, 8.9.0, or 8.12.0. 46 projects were compatible with version 8.16.0, and 36 projects with version 8.20.0, *i.e.*, the latest version of Rocq as of conducting our experiments. Coq 8.16.0 is the first version of Rocq that includes Coq-LSP, so we had to use version 8.16.0 or newer. We chose Coq 8.16.0, because it was compatible with most number of projects. Unfortunately, the older version of Coq-LSP, shipped with Coq 8.16.0, occasionally crashed with a “segmentation fault” error, when we used it through CoqPyt, so we ended up with 40 working Rocq projects.

Columns “Project Name” and “Project Size” of Table 1 list all the projects in our benchmark and their sizes. We have reported the project sizes both in terms of number of `.v` files in the project (column “#Files”) and total lines of code (column “LoC”). Our benchmark contains a diverse set of Rocq projects, representative of the real-world Rocq projects. The projects are of varying sizes from 53 to 29,260 lines of code, and they are from various domains: from set theory to distributed system verification to geometry.

5.2 Results

We ran `clone-finder` on each of our benchmark projects. The last four columns of Table 1 report our experimental results in terms of number clones, *i.e.*, the number of α -equivalent goal pairs with minimum proof size of 5 lines of Ltac code, as well as the time, in seconds. We make three time measurements: (1) under the column “Build,” we report the time for running Rocq on all the project files to build and generate `.vo` files; (2) under the column “clone-finder Total,” we report the end-to-end time for running `clone-finder`, including the time Rocq needs to load and type-check the files; and (3) under the column “Analysis,” we report the time `clone-finder` needs to parse the goals, generalize and deduplicate them, and find α -equivalent ones. Studying clones with different proof sizes is left as a future work.

In 20 out of 40 projects, `clone-finder` finds at least one clone. The average number of goal clones per project is 27.73 (with median being 0.5). Larger projects tend to have more instances of redundancy in them. By inspecting the clones and their proofs, we observed 3 major types of proofs: (1) identical or near identical proofs, where the proofs are almost the same, except the variable names have changed; (2) generalized proofs, where one proof proves a more general form of the goal; (3) entirely different proofs, where new proofs have been invented in each case the goal is observed.

Figure 3 shows an example of a pair of α -equivalent goals and their identical proofs. We do not have information about how the authors have written these proofs. Specifically, we are not sure whether the author have copied and pasted the proofs or they have rewritten the proofs in each case. Either way, we believe that this kind of redundancy is the least expensive one, as no new theorem or a generalization thereof needs to be created.

12:12 Automatic Goal Clone Detection in Rocq

■ **Table 1** Benchmark of Rocq projects and the result of applying clone-finder on each project.

Project Name	Project Size		# Clones	Time (s)		
	# Files	LoC		Build	clone-finder Total	Analysis
AlmostFull	11	3,625	102	8.563	1962.643	27.661
Angles	6	3,633	25	4.356	79.379	50.821
Autosubst	8	1,553	0	3.484	104.820	0.511
bbv	16	9,027	6	13.460	3137.301	93.807
CDF	13	8,558	33	14.127	3502.113	21.409
Chapar	10	20,371	447	303.367	2896.301	223.292
Checker	2	141	0	0.950	11.309	0.016
ConstructiveGeometry	7	1,453	0	2.123	21.533	0.204
Coqoban	2	6,398	0	2.286	16.486	0.947
DepMap	6	1,259	0	6.701	1538.239	5.151
DomainTheory	6	1,223	1	3.472	96.321	5.981
ExtLib	118	10,304	3	29.696	2860.590	9.163
FreeGroups	1	654	0	1.166	71.283	0.785
FunctionsInZFC	1	6,294	0	1.414	18.383	0.001
Game	15	1,345	1	2.950	101.722	1.158
Groups	1	306	0	0.850	8.788	0.089
GroupTheory	12	1,661	2	3.793	843.374	1.891
Hedges	1	3,265	8	2.775	164.673	32.239
HighSchoolGeometry	72	29,260	218	185.082	29773.385	1062.286
Huffman	22	6,791	9	15.160	3112.586	29.449
IZF	9	2,419	3	2.207	23.045	1.169
JML	30	20,009	27	23.594	5292.184	48.636
Lambda	12	2,122	0	5.559	1374.476	1.899
LibHyps	8	2,957	0	2.013	1111.587	0.000
mathcomp.bigenough	1	122	0	1.272	185.391	0.000
mathcomp.finmap	2	5,029	0	19.181	1204.847	3.433
mathcomp.zify	4	1,583	0	23.888	3616.383	0.025
MiniML	1	1,425	3	1.958	24.486	6.717
NotationGallery	1	53	0	0.120	93.005	0.000
OtwayRees	21	1,401	16	5.850	119.969	8.015
parseque	14	833	0	4.811	749.475	0.000
Ramsey	1	172	0	0.603	50.528	0.000
RecordUpdate	3	131	0	0.947	11.400	0.000
RSA	5	1,654	0	3.991	1010.601	13.068
Schroeder	5	640	0	1.503	76.689	0.573
Stalmarck	38	14,108	78	30.861	7819.962	134.503
Subst	20	7,506	82	11.001	132.391	9.668
WeakUpTo	10	2,334	7	3.433	26.097	2.284
ZFC	11	4,336	38	4.538	47.417	15.437
ZF	15	7,977	0	5.015	42.641	0.000

Figure 4 shows an example of the second type of redundancy, wherein one goal is in fact a generalization of the other. In the first case, the local variable eH is bound to a specific value of type U , while in the second case, the author proves the same theorem for all values of type U . In such cases, it would be possible to factor out the more general goal as an independent lemma, which could also be useful for future developments of the system.

```

1      (*Goal 1:*) step_star (init ?x) ?x0 ?x1 /\ In ?x2 (messages ?x1)
2      (*Proof 1:*)
3      split_all.
4      eassumption.
5      subv si.
6      apply in_app_iff.
7      right.
8      apply in_eq.
9
10     (*Goal 2:*) step_star (init ?x) ?x0 ?x1 /\ In ?x2 (messages ?x1)
11     (*Proof 2:*)
12     split_all.
13     eassumption.
14     subv si.
15     apply in_app_iff.
16     right.
17     apply in_eq.

```

■ **Figure 3** Duplicate proofs example: Example of two goals from Rocq project Chapar where α -equivalent goals show up in two different theorems `KVSA1g1Cause0b1.algrec_step` and `KVSA1g1Cause0b1.cause_rec` and identical proofs have been used.

Lastly, Figure 5 shows an example of the third kind of redundancy, *i.e.*, entirely different proofs have been proposed for α -equivalent goals. This could have educational value, *e.g.*, proving same goals in different ways, or the authors have proved these goals at different times, or different people have proved the goals, and a refactoring opportunity was missed.

Another observation that we can make from Table 1 concerns the time measurements. We observe that, on average, it takes 1,833.345 seconds for `clone-finder` to analyze a Rocq project in our benchmark to find clones. We further observed that 94.71% of this end-to-end time is from using CoqPyt/Coq-LSP to execute the Rocq files step by step, as the average time required to find clones after type checking the files is only 45.31 seconds (with median being 2.091 seconds). This means that, compared to normal build (normal build time is reported under “Build” column in Table 1), on average, there is a 122.4X increase in run time. This significant overhead is because CoqPyt loads Rocq script files individually, rather than loading the entire project as a batch, which calls for loading and checking all the dependencies of the script file each time we load Rocq script. We are still investigating if this issue can be resolved by only modifying CoqPyt so that its `ProofFile` class can accept a Rocq project to load, and cache the common dependencies between the files in the project and avoid loading and checking them from scratch, or if a modification to Coq-LSP is also needed. Another way to mitigate this overhead would be through incorporating `clone-finder` as a Rocq plugin that directly retrieves the AST for goals and contexts from Rocq platform itself. We save these engineering enhancements as a future extension of this work. The current version of `clone-finder` tool stores partial results in file and avoids invoking Coq-LSP whenever the cached results are already available.

6 Threats to Validity

Like all empirical studies, our findings are subject to threats to validity. In this section we discuss some of these threats and how we mitigate them.

We had to limit our experiments to the set of Rocq projects that were compatible with Coq 8.16.0, which slashed the size of our benchmark by 86 projects. Additionally, the Coq-LSP version shipped with Coq 8.16.0 occasionally crashed, leading to a reduced set of 40 working Rocq projects out of the 46 projects compatible with Coq 8.16.0. Despite these cuts in the size of the benchmark, the Rocq projects are of varying sizes and domains, giving us some level of confidence about their representativeness. Nevertheless, this reduction could introduce bias by excluding certain types of projects or proofs from the analysis, potentially affecting the representativeness of the results. To mitigate this threat, we will make `clone-finder` open-source so that research community can study it on larger available Rocq dataset.

```

1      (*Goal 1: *) In U H eH -> Setsubgroup U H Gr
2      (*Proof 1: *)
3      intro H'.
4      apply T_1_6_2 with (witness := witness); auto with sets.
5      red in |- *; intros a b H'0 H'1; try assumption.
6      lapply (h2 a (inv b)); [ intro H'5; lapply H'5; [ intro H'6; generalize H'6; clear H'5 | clear H'5 ] | idtac ];
7      auto with sets.
8      rewrite <- (inv_involution' b); auto with sets.
9      red in |- *; intros a H'0; try assumption.
10     rewrite <- (G2b' (inv a)).
11     apply h2; auto with sets.
12     rewrite <- (G3b' witness); auto with sets.
13
14     (*Goal 2: *) forall x : U, In U H x -> Setsubgroup U H Gr
15     (*Proof 2: *)
16     intros witness inH.
17     apply T_1_6_2 with (witness := witness); trivial.
18     red in |- *.
19     intros a H'.
20     cut (exists n : nat, inv a = phi a n).
21     intro H'0; elim H'0; intros n E; rewrite E; clear H'0.
22     apply positive_powers; trivial.
23     cut (inv a = phi a m).
24     intro H'0; rewrite H'0.
25     exists m; trivial.
26     symmetry in |- *.
27     apply powers_repeat with (n := r); trivial.
28     apply H_included_in_G; auto.

```

■ **Figure 4** More general proof reinvented: Example of two goals from Rocq project GroupTheory where α -equivalent goals show up in two different theorems T_1_6_3 and T_1_6_4. The second proof is a more general form of the first one, and the proofs seem to be reinvented for each goal.

clone-finder looks for α -equivalent goals in Rocq projects. In other words, in this paper, we only consider syntactic clones. Our results are not generalizable to the other major type of clones. Specifically, the projects in our benchmark could contain semantic clones, *e.g.*, closely related goals that can be generalized into a single more general goal, or even universally quantified goal pairs with non-dependent quantified variables that are different only in the order of their quantified variables. We save this interesting topic for a future work (see §8).

The basic idea behind Algorithm 1 is inspired from [42]. Similar algorithm has been used in other works as well [32, 35]. This style of constructing proof trees has the drawback of reporting inaccurate results when facing with compound tactics, tactics that resolves more than one goal, and focus-shifting tactics. Some authors [35] have explicitly mentioned that such commands would break the algorithm, while others [42] have desugared or otherwise filtered out the proofs containing such tactics. In this work, we opted for not filtering out proofs, as in our application, missing a few clone cases or reporting inaccurate proof boundaries for a goal is not harmful.

Last but not least, the primary metrics used to evaluate clone-finder are the number of clones detected and the time taken for analysis. While these metrics provide significant insights into the performance of the proposed technique, they may not fully capture the practical significance or impact of the detected clones. Additional metrics, such as the effort required to refactor or eliminate the clones will be studied in a future work.

7 Related Work

The works by Ghanouchi [18] and Vinan and Hupel [40] that introduce two approaches for clone detection in Isabelle/HOL theories are closest to clone-finder. Ghanouchi [18] applies fuzzy token matching to detect clones in Isabelle/HOL theory files. This technique extends previous token-based approaches by considering near-miss clones through similarity-based token matching. Vinan and Hupel [40], on the other hand, adapt the ConQAT framework, a well-known clone detection tool, to detect proof clones in Isabelle/HOL theories. This technique extracts structured information from Isabelle document markup and performs syntactic clone detection using ConQAT’s text-based similarity analysis. Unlike these two techniques, clone-finder is not intended to find clones in proof scripts themselves. Instead,

```

1      (*Goal 1:*) orthogonal (vec H D) (vec H C)
2      (*Proof 1:*)
3      apply ortho_sym.
4      rewrite H11.
5      Vreplace (vec H A) (mult_PP (-1) (vec A H)).
6      rewrite H12.
7      Vreplace (mult_PP k (mult_PP (-1) (mult_PP k' (vec A B)))) (mult_PP (- (k * k')) (vec A B)).
8      auto with geo.
9
10     (*Goal 2:*) orthogonal (vec H D) (vec H A)
11     (*Proof 2:*)
12     apply ortho_sym.
13     replace (vec H A) with (mult_PP (- k) (vec A B)).
14     auto with geo.
15     Vreplace (vec H A) (mult_PP (-1) (vec A H)).
16     rewrite H8; Ringvec.

```

■ **Figure 5** Reinvented proofs example: Example of two goals from Rocq project `HighschoolGeometry` where α -equivalent goals show up in two different theorems `intersection_cercle_droite` and `intersection2_cercle_droite`, different proofs seem to be reinvented.

it is designed to detect goals that could be addressed once and for all. In other words `clone-finder` leverages notions of α -equivalence of goals, and proof irrelevance [4], to detect redundant proof efforts. One major benefit of `clone-finder`'s clone detection approach, driven by α -equivalence notion, is that it can be sound and complete, but the aforementioned similarity-based heuristics may result in false positives/negatives.

Goal clone detection aligns with *proof reuse* in proof engineering. “Large-scale proof development may involve redundant efforts that can be time-consuming. Proof reuse addresses this by repurposing existing proofs as much as possible, minimizing the amount of redundant work that proof engineers must do” [33]. Existing proof reuse techniques mainly focus on proof generalization [11, 19, 5, 3, 34, 21]. To the best of our knowledge, `clone-finder` is the first technique on the topic of goal clone detection that realizes proof reuse by identifying α -equivalent goals that can be extracted as independent lemmas and reused. As such lemma extraction tools for Rocq, *e.g.*, `company-coq` [32] and `CoqPIE` [35], can be used alongside `clone-finder`.

Finding α -equivalent goals in Rocq proofs is directly related to the software engineering problem Type 1, *i.e.*, exact clones, and Type 2, rename clones, clone detection. There is a rich body of knowledge developed for clone detection for various procedural programming languages [28, 36, 29, 41] and even Java bytecode [25, 43]. Being language dependent, none of these techniques can be directly applied to goal clone detection. Additionally, functional programming defines α -equivalence, which is a theoretically sound notion of syntactic clones, *i.e.*, Type 1 and Type 2 combined, that can be leveraged in the context of functional programming, in general, and Rocq, in particular, without resorting to potentially inaccurate heuristics proposed in the past.

8 Conclusion and Future Work

We design, implement, and evaluate a technique and a tool, named `clone-finder`, for detecting goal clones, *i.e.*, instances of α -equivalent goals that are proved multiple times within a Rocq project. Our evaluation of `clone-finder` on 40 real-world Rocq projects reveals that half of the projects (20 out of 40) contained at least one goal clone. The detected redundancies fell into three primary categories: (1) exact goal duplication with identical or near-identical proofs, *e.g.*, structurally identical proofs with different variable names; (2) α -equivalent goals with generalized proofs, where one proof subsumes the other; and (3) α -equivalent goals with entirely different proofs. These findings highlight a significant opportunity for improving proof maintainability and reusability in large-scale Rocq developments.

Performance analysis of clone-finder provided empirical evidence that it is a lightweight system, with an average runtime of 45.31 seconds per project, making it a practical tool for daily proof engineering tasks. Additionally, in the process of implementing clone-finder, we contributed an enhancement to CoqPyt, allowing for handling of physical-to-logical path mappings in Rocq projects.

Our results suggest that redundancy in proof engineering is a widespread issue and that automation could help mitigate wasted effort. Moving forward, we envision extending clone-finder to detect semantic clones with the help of large language models, integrating it directly as a Rocq plugin for improved efficiency, and exploring automated lemma extraction to further reduce redundant proof efforts. We are also planning to incorporate an η -equivalence detection mechanism in clone-finder.

References

- 1 Eyad Alkassar, Mark A. Hillebrand, Dirk C. Leinenbach, Norbert W. Schirmer, Artem Starostin, and Alexandra Tsyban. Balancing the load. *Journal of Automated Reasoning*, 42(2):389–454, April 2009. doi:10.1007/s10817-009-9123-z.
- 2 Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. Concert: a smart contract certification framework in coq. In Jasmin Blanchette and Catalin Hritcu, editors, *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020*, pages 215–228. ACM, 2020. doi:10.1145/3372885.3373829.
- 3 Gilles Barthe and Olivier Pons. Type isomorphisms and proof reuse in dependent type theory. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures, 4th International Conference, FOSSACS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2030 of *Lecture Notes in Computer Science*, pages 57–71. Springer, 2001. doi:10.1007/3-540-45315-6_4.
- 4 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. doi:10.1007/978-3-662-07964-5.
- 5 Alex J. Best. Automatically generalizing theorems using typeclasses (short paper). In Jasmin Blanchette, James H. Davenport, Peter Koepke, Michael Kohlhase, Andrea Kohlhase, Adam Naumowicz, Dennis Müller, Yasmine Sharoda, and Claudio Sacerdoti Coen, editors, *Joint Proceedings of the FMM, FVPS, MathUI, NatFoM, and OpenMath Workshops, Doctoral Program, and Work in Progress at the Conference on Intelligent Computer Mathematics 2021 co-located with the 14th Conference on Intelligent Computer Mathematics (CICM 2021), Virtual Event, Timisoara, Romania, July 26 - 31, 2021*, volume 3377 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2021. URL: <https://ceur-ws.org/Vol-3377/fmm12.pdf>.
- 6 Timothy Bourke, Matthias Daum, Gerwin Klein, and Rafal Kolanski. Challenges and experiences in managing large-scale proofs. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel, and Volker Sorge, editors, *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proceedings*, volume 7362 of *Lecture Notes in Computer Science*, pages 32–48. Springer, 2012. doi:10.1007/978-3-642-31374-5_3.
- 7 Pedro Carrott, Nuno Saavedra, Kyle Thompson, Sorin Lerner, João F. Ferreira, and Emily First. Coqpyt: Proof navigation in python in the era of llms. In Marcelo d'Amorim, editor, *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 637–641. ACM, 2024. doi:10.1145/3663529.3663814.

- 8 Coq-Elpi Team. Coq-Elpi: Coq plugin embedding elpi, 2025. Accessed: 04/22. URL: <https://github.com/LPCIC/coq-elpi>.
- 9 Coq-LSP Team. Coq-LSP: Visual Studio Code Extension and Language Server Protocol for Rocq / Coq, 2025. Accessed: 04/22. URL: <https://github.com/ejgallecoq/coq-lsp>.
- 10 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- 11 Régis Curien. *Outils pour la preuve par analogie*. Theses, Université Henri Poincaré - Nancy 1, January 1995. URL: <https://hal.univ-lorraine.fr/tel-01748604>.
- 12 David Delahaye. A tactic language for the system coq. In Michel Parigot and Andrei Voronkov, editors, *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, volume 1955 of *Lecture Notes in Computer Science*, pages 85–95. Springer, 2000. doi:10.1007/3-540-44404-1_7.
- 13 Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic: With proofs, without compromises. *ACM SIGOPS Oper. Syst. Rev.*, 54(1):23–30, 2020. doi:10.1145/3421473.3421477.
- 14 Emily First and Yuriy Brun. Diversity-driven automated formal verification. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pages 1–13. ACM, 2022. doi:10.1145/3510003.3510138.
- 15 Emily First, Yuriy Brun, and Arjun Guha. Tactok: semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA):231:1–231:31, 2020. doi:10.1145/3428299.
- 16 Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In Satish Chandra, Kelly Blincoe, and Paolo Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1229–1241. ACM, 2023. doi:10.1145/3611643.3616243.
- 17 Ali Ghanbari. Automatic Goal Clone Detection in Rocq, 2025. Accessed: 04/22. URL: <https://github.com/ali-ghanbari/clone-finder>.
- 18 Seifeddine Ghanouchi. Code clone detection in isabelle using token stream similarity. Technical report, Technical University of Munich, 2023. Bachelor's Thesis.
- 19 Robert W Hasker and Uday S Reddy. Generalization at higher types. In *Proceedings of the Workshop on the λ Prolog Programming Language*, pages 257–271, 1992.
- 20 Inria. The coq proof assistant documentation, 2025. Accessed: 04/22. URL: <https://rocq-prover.org/doc/V9.0.0/refman/index.html>.
- 21 Einar Broch Johnsen and Christoph Lüth. Theorem reuse by proof term transformation. In Konrad Slind, Annette Bunker, and Ganesh Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*, pages 152–167. Springer, 2004. doi:10.1007/978-3-540-30142-4_12.
- 22 Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28:e20, 2018. doi:10.1017/S0956796818000151.
- 23 Elmar Jürgens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 485–495. IEEE, 2009. doi:10.1109/ICSE.2009.5070547.
- 24 Cory Kapser and Michael W. Godfrey. "cloning considered harmful" considered harmful: patterns of cloning in software. *Empir. Softw. Eng.*, 13(6):645–692, 2008. doi:10.1007/S10664-008-9076-6.
- 25 Iman Keivanloo, Chanchal K. Roy, and Juergen Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In James R. Cordy, Katsuro Inoue, Rainer Koschke, Jens Krinke, and Chanchal K. Roy, editors, *Proceeding of the 6th International Workshop on Software Clones, IWSC 2012, Zurich, Switzerland, June 4, 2012*, pages 36–42. IEEE Computer Society, 2012. doi:10.1109/IWSC.2012.6227864.

- 26 Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an operating-system kernel. *Commun. ACM*, 53(6):107–115, 2010. doi:10.1145/1743546.1743574.
- 27 Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.
- 28 Micheline Bénédicte Moumoula, Abdoul Kader Kaboré, Jacques Klein, and Tegawendé F. Bis-syandé. Large language models for cross-language code clone detection. *CoRR*, abs/2408.04430, 2024. doi:10.48550/arXiv.2408.04430.
- 29 Morteza Zakeri Nasrabadi, Saeed Parsa, Mohammad Ramezani, Chanchal Roy, and Masoud Ekhtiarzadeh. A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges. *J. Syst. Softw.*, 204:111796, 2023. doi:10.1016/J.JSS.2023.111796.
- 30 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.
- 31 Christine Paulin-Mohring. Inductive definitions in the system coq - rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993. doi:10.1007/BFB0037116.
- 32 Clément Pit-Claudel and Pierre Courtieu. Company-coq: Taking proof general one step closer to a real ide. In *CoqPL'16: The Second International Workshop on Coq for PL*, January 2016. doi:10.5281/zenodo.44331.
- 33 Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. QED at large: A survey of engineering of formally verified software. *CoRR*, abs/2003.06458, 2020. arXiv:2003.06458.
- 34 Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. Ornaments for proof reuse in coq. In John Harrison, John O’Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving, ITP 2019, September 9-12, 2019, Portland, OR, USA*, volume 141 of *LIPICs*, pages 26:1–26:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPICS.ITP.2019.26.
- 35 Kenneth Roe and Scott F. Smith. Coqpie: An IDE aimed at improving proof development productivity - (rough diamond). In Jasmin Christian Blanchette and Stephan Merz, editors, *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, volume 9807 of *Lecture Notes in Computer Science*, pages 491–499. Springer, 2016. doi:10.1007/978-3-319-43144-4_32.
- 36 Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen’s School of computing TR*, 541(115):64–68, 2007.
- 37 Alex Sanchez-Stern, Yousef Alhessi, Lawrence K. Saul, and Sorin Lerner. Generating correctness proofs with neural networks. In Koushik Sen and Mayur Naik, editors, *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2020, London, UK, June 15, 2020*, pages 1–10. ACM, 2020. doi:10.1145/3394450.3397466.
- 38 Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. Passport: Improving automated formal verification using identifiers. *ACM Trans. Program. Lang. Syst.*, 45(2):12:1–12:30, 2023. doi:10.1145/3593374.
- 39 Paula Severi and Erik Poll. Pure type systems with definitions. In Anil Nerode and Yuri V. Matiyasevich, editors, *Logical Foundations of Computer Science, Third International Symposium, LFCS’94, St. Petersburg, Russia, July 11-14, 1994, Proceedings*, volume 813 of *Lecture Notes in Computer Science*, pages 316–328. Springer, 1994. doi:10.1007/3-540-58140-5_30.

- 40 Dominik Vinan and Lars Hupel. Clone detection in isabelle theories. Technical report, Technical University of Munich, 2016.
- 41 Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 87–98. ACM, 2016. doi:10.1145/2970276.2970326.
- 42 Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6984–6994. PMLR, 2019. URL: <http://proceedings.mlr.press/v97/yang19a.html>.
- 43 Dongjin Yu, Jie Wang, Qing Wu, Jiazha Yang, Jiaojiao Wang, Wei Yang, and Wei Yan. Detecting java code clones with multi-granularities based on bytecode. In Sorel Reisman, Sheikh Iqbal Ahamed, Claudio Demartini, Thomas M. Conte, Ling Liu, William R. Claycomb, Motonori Nakamura, Edmundo Tovar, Stelvio Cimato, Chung-Horng Lung, Hiroki Takakura, Ji-Jiang Yang, Toyokazu Akiyama, Zhiyong Zhang, and Md. Kamrul Hasan, editors, *41st IEEE Annual Computer Software and Applications Conference, COMPSAC 2017, Turin, Italy, July 4-8, 2017. Volume 1*, pages 317–326. IEEE Computer Society, 2017. doi:10.1109/COMPSAC.2017.104.