ROOSTERIZE: Suggesting Lemma Names for Coq Verification Projects Using Deep Learning

Pengyu Nie*, Karl Palmskog†, Junyi Jessy Li*, Milos Gligoric*
pynie@utexas.edu, palmskog@kth.se, jessy@austin.utexas.edu, gligoric@utexas.edu
* *The University of Texas at Austin*, Austin, TX, USA † *KTH Royal Institute of Technology*, Stockholm, Sweden

***Abstract*—Naming conventions are an important concern in large verification projects using proof assistants, such as Coq. In particular, lemma names are used by proof engineers to effectively understand and modify Coq code. However, providing accurate and informative lemma names is a complex task, which is currently often carried out manually. Even when lemma naming is automated using rule-based tools, generated names may fail to adhere to important conventions not specified explicitly. We demonstrate a toolchain, dubbed ROOSTERIZE, which automatically suggests lemma names in Coq projects. ROOSTERIZE leverages a neural network model trained on existing Coq code, thus avoiding manual specification of naming conventions. To allow proof engineers to conveniently access suggestions from ROOSTERIZE during Coq project development, we integrated the toolchain into the popular Visual Studio Code editor. Our evaluation shows that ROOSTERIZE substantially outperforms strong baselines for suggesting lemma names and is useful in practice. The demo video for ROOSTERIZE can be viewed at: https://youtu.be/HZ5ac7Q14rc.**

***Index Terms*—Coq, lemma names, neural networks**

I. INTRODUCTION

In large software projects with many contributors, names of methods and classes are important for code comprehension and modification. Open source projects often document their naming conventions carefully, impose them on proposed contributions, and willingly accept naming fixes [1].

The Coq proof assistant [2] is increasingly used to develop trustworthy software systems, e.g., compilers [3] and distributed systems [4]. As such verification projects grow in scope and size, naming conventions become an important concern. In particular, proof engineers use *lemma names* to effectively understand and modify code [5].

In contrast to method names in Java-like languages, which tend to use camel case and regular English words (e.g., `openServerConnection`), Coq lemma names often mix camel case and underscores with heavily abbreviated terminology from logic and advanced mathematics, which makes the naming task more difficult. For example, in the Mathematical Components (MathComp) Coq library, the lemma name `extprod_mulgA` is used to express "associativity of multiplication operations in external product groups", i.e., a property of abstract algebra. This meaning is obtained by first decomposing the name into `extprod`, `mul`, `g`, and `A`, and then consulting the MathComp naming conventions [6].

Currently, documentation and enforcement of lemma naming conventions in Coq projects is largely a manual process. While some aspects of naming conventions can be captured by rule-based tools, specification of rules is tedious and often *incomplete*. Moreover, most large Coq projects use mutually incompatible lemma naming schemes.

We present ROOSTERIZE, a toolchain which automatically suggests Coq lemma names. ROOSTERIZE learns naming conventions by leveraging neural networks trained on existing Coq code. The deep learning and suggestion processes use multiple representations of lemma statements, including syntax trees and Coq *kernel trees* (also called *elaborated terms*) [7]. In essence, ROOSTERIZE consists of (1) a set of components written in OCaml that interact with Coq or directly process information extracted from Coq, and (2) a set of components written in Python that perform name learning and generation. The first set of components is based on the SerAPI library [8] for serialization of Coq data, while the second set of components is based on the PyTorch deep learning framework [9] and the OpenNMT library [10].

The core of ROOSTERIZE is command-line based. Although valuable, this does not provide a convenient interface for proof engineers as they are stating and proving new Coq lemmas. Hence, we integrated the toolchain into Visual Studio Code (VSCode) [11], a popular editor for Coq source code.

We evaluated an earlier version of ROOSTERIZE using a corpus derived from the MathComp family of Coq projects, finding that the toolchain significantly outperforms strong baselines on automatic metrics [7]. Moreover, we found encouraging results in a qualitative case study where the maintainer of a medium-sized Coq project manually evaluated over 150 name suggestions generated by ROOSTERIZE.

The earlier toolchain version provided few conveniences beyond basic name suggestions via the command line, and did not include any editor integration. In addition to the novel integration with VSCode, the toolchain version presented here provides a significantly more automated installation process and supports system-level configuration of Coq project and name suggestion parameters, making it suitable for wider use by proof engineers.

Our code, documentation, and pre-trained models are publicly available on GitHub:
https://github.com/EngineeringSoftware/roosterize.

II. TECHNIQUE AND IMPLEMENTATION

In this section, we explain the workflow of the ROOSTERIZE toolchain, and then briefly describe our neural network model for lemma name generation.

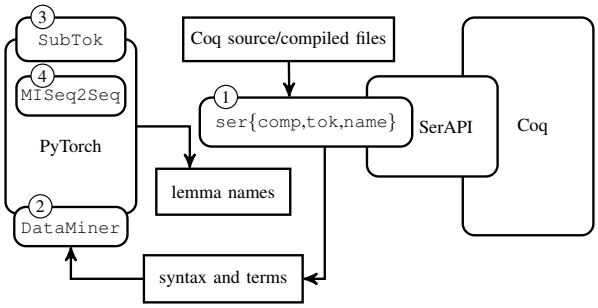

Fig. 1: Low-level workflow of ROOSTERIZE.

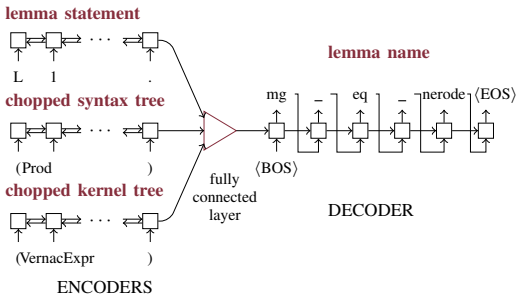

Fig. 2: Neural architecture of lemma name generation model in ROOSTERIZE, exemplified for the name `mg_eq_nerode`.

A. Toolchain Workflow

Fig. 1 illustrates the low-level workflow of the ROOSTERIZE toolchain. (1) We use the SerAPI library [8] for extracting data from Coq files, using three programs: `sertok` for extracting tokens, `sercomp` for extracting the syntax trees, and `sername` for extracting kernel trees. Syntax trees are Coq's internal representations of source code elements, including lemmas, during the parsing phase. Kernel trees are Coq's internal representations of statements and functions during proof checking, and contain rich information relevant for lemma naming. (2) `DataMiner` orchestrates these programs to obtain all lemmas in the given Coq files and their names, lemma statements, and syntax and kernel trees. (3) `SubTok` sub-tokenizes the inputs for the neural network model. (4) `MISeq2Seq` is the multi-input neural network model for lemma name generation (Section II-B). The model is implemented in the popular deep learning framework PyTorch [9], and is based on the OpenNMT library [10].

Users interact with ROOSTERIZE by using its command-line interface or the VSCode extension. We use the Language Server Protocol (LSP) [12] to connect the server (the core of ROOSTERIZE including data extraction scripts and the lemma name generation model) with the client (the VSCode extension). This simplifies future integration with other editors that also support LSP, e.g., Emacs.

B. Neural Network Model for Generating Lemma Names

We consider lemma name generation with an *encoder-decoder* mindset, and use sequence-to-sequence (SEQ2SEQ) neural architectures specifically designed for transduction tasks [13]. Fig. 2 illustrates the architecture of our model [7]. The encoders are Recurrent Neural Networks (RNNs) that learn a deep semantic representation of a given lemma statement from its tokens, syntax tree, and kernel tree. The model can be configured to use any combination of the three encoders. The decoder is another RNN that generates the descriptive lemma name based on the input deep semantic representation. We equipped the decoder RNN with attention mechanism [14] and copy mechanism [15] to improve the generation accuracy.

All the inputs and the output are sequences of sub-tokens; the sub-tokens of inputs are obtained using a sub-tokenizer, and the sub-tokens of the output are concatenated to form the generated lemma name. We implemented the sub-tokenizer based on the conventions outlined by MathComp developers [6] (e.g., the lemma name `extprod_mulgA` should be sub-tokenized to `extprod`, `_`, `mul`, `g`, and `A`). Because syntax and kernel trees can be large, we implemented *chopping* heuristics to remove the parts irrelevant for generating lemma names before feeding them to the encoders. Our heuristics essentially: (1) replace the fully qualified name sub-trees with only the last component of the name; (2) remove the line number information from sub-trees; (3) extract the singletons, i.e., non-leaf nodes that have only one child.

III. TOOL INSTALLATION

ROOSTERIZE currently supports macOS and Linux-based operating systems. The first installation step is to download the ROOSTERIZE repository:

```
$ git clone \
    https://github.com/EngineeringSoftware/roosterize
$ cd roosterize && git checkout v1.1.0+8.10.2
```

Required software and libraries. ROOSTERIZE depends on two sets of software and libraries: (1) OCaml, Coq, and SerAPI; (2) PyTorch and other Python libraries.

To install OCaml (4.07.1), Coq (8.10.2) and SerAPI (0.7.1), we recommend using the OCaml-based package-management system OPAM [16] version 2.0.7 or later:

```
$ opam switch create roosterize 4.07.1
$ opam switch roosterize && eval $(opam env)
$ opam update
$ opam pin add coq 8.10.2
$ opam pin add coq-serapi 8.10.0+0.7.1
```

To install PyTorch and other Python libraries, we recommend using the package-management system Conda [17]. The installation script may be different depending on the operating system and whether to use GPU or not. For example, on Linux, to use CPU only:

```
$ conda env create --name roosterize \
                   --file conda-envs/cpu.yml
$ conda activate roosterize
```

After installing these required software and libraries, users can use ROOSTERIZE via its command-line interface.

VSCode extension. The ROOSTERIZE VSCode extension can be installed easily from VSCode marketplace: launch "VS Code Quick Open" (Ctrl+P), paste the following command:

```
ext install EngineeringSoftware.roosterize-vscode
```

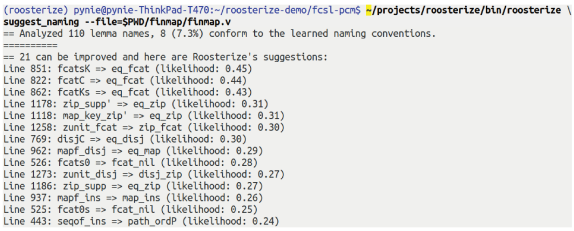

Fig. 3: Screenshot of using ROOSTERIZE from command line.

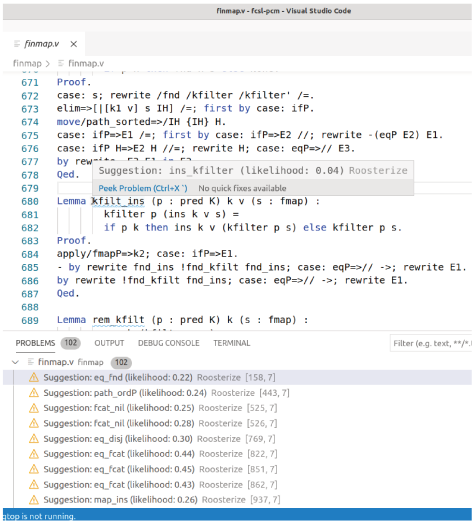

Fig. 4: Screenshot of using ROOSTERIZE from VSCode.

Then, users should configure the path to ROOSTERIZE executable file (`./bin/roosterize`) using the following steps: open "Settings" (Ctrl+,), search for the entry "ROOSTERIZE: Bin Path", and fill in the path to ROOSTERIZE executable file.

IV. TOOL USAGE

In this section, we describe the way our tool can be invoked from command line and from VSCode.

A. Command Line

After installation, users can launch ROOSTERIZE via the executable file `./bin/roosterize` (in short, `roosterize`). We focus on the main usage of ROOSTERIZE—suggesting lemma names for Coq projects. For other usages, e.g., training models, users can refer to the help included in ROOSTERIZE:

```
$ roosterize --help
```

Users should first obtain a model, e.g., by downloading a pre-trained model. The following command downloads the model we pre-trained on our MathComp corpus [7]:

```
$ roosterize download_global_model
```

Applying ROOSTERIZE to a Coq project requires (1) a `_CoqProject` file in the project root directory in the format used by the `coq_makefile` tool [18], and (2) that the project source code has been compiled. If a user specified the compilation command in the `.roosterizerc` configuration file at the root directory of the project, ROOSTERIZE will automatically compile the project before suggesting lemma names. ROOSTERIZE can suggest lemma names for one Coq file at a time. For example, running the following commands downloads the Coq project FCSL PCM at Git revision eef4503, prepares a `.roosterizerc` configuration file, and suggests lemma names for the `finmap.v` file in the project:

```
$ git clone \
   https://github.com/imdea-software/fcsl-pcm
$ git checkout eef4503
$ echo "compile_cmd: make -j8" > ./.roosterizerc
$ roosterize suggest_naming \
             --file=$PWD/finmap/finmap.v
```

In the last command, ROOSTERIZE uses SerAPI to parse `finmap.v` and extracts all lemmas, then uses the lemma names suggestion model to generate top k (default $k = 5$) likely lemma names for each lemma, and then compares the generated lemma names with the original lemma names, and finally prints a report to suggest potential lemma names changes. Fig. 3 shows the top part of the report generated for `finmap.v` (full report available at: https://tinyurl.com/yy6l8fbq).

B. VSCode

Users should first open the Coq files they want to analyze. The steps to obtain the lemma names suggestions for them are: (1) open "Command Palettes" (Ctrl+Shift+P), and (2) choose "Roosterize: Suggest Naming (for all .v files)". After ROOSTERIZE produces the suggestions, the lemma names that do not conform to the conventions are underlined, and users can hover the mouse pointer over that underlined name to view ROOSTERIZE's suggestion in a tooltip. Users can also view all suggestions in the "Problems" tab. Fig. 4 shows a screenshot of this step.

V. EVALUATION

A. Quantitative

Coq projects under study. We selected four large Coq projects from the MathComp family: math-comp, finmap, fourcolor, and odd-order. The projects have a total of 164k lines of code, 187 files, and 11,266 lemmas. More information of these projects can be found in our corpus [19] ("Tier 1" part). Following standard deep learning practice, we randomly split the projects' files into training, validation, and testing sets which contain 80%, 10%, 10% of the files, which is 8,861, 1,085, 1,320 lemmas, respectively; the lemmas from a same file are assigned to only one of the sets.

Results. We trained various configurations of our model. In this paper, we focus on five configurations that use different inputs: Stmt+ChopKnlTree+ChopSynTree, Stmt+ChopKnlTree, Stmt+ChopSynTree, ChopKnlTree+ChopSynTree, and only Stmt (where Stmt = lemma statement, ChopSynTree = chopped syntax tree, ChopKnlTree = chopped kernel tree). We also compare our model with a retrieval-based baseline.

We evaluate each model by applying it on the testing set and measure the average similarity between the generated names and the expected names (as written by developers), using four automatic metrics: BLEU [20], fragment accuracy [7], top-1 accuracy, and top-5 accuracy.

TABLE I: Results of ROOSTERIZE Models.

Model	BLEU	Frag.Acc.	Top1	Top5
Stmt+ChopKnlTree+ChopSynTree	45.4	22.2%	7.5%	16.5%
Stmt+ChopKnlTree	**47.2**	**24.9%**	**9.6%**	**18.0%**
Stmt+ChopSynTree	37.7	18.1%	6.1%	10.6%
ChopKnlTree+ChopSynTree	45.4	22.9%	7.6%	15.3%
Stmt	38.9	19.4%	6.9%	11.6%
Retrieval-based	28.3	10.0%	0.2%	0.3%

Table I shows the results. We observed that Stmt +ChopKnlTree achieved the best performance and substantially outperforms the retrieval-based baseline. This shows the importance of using Coq's internal structures. Lemma statement and syntax tree do not work well together primarily because the two representations contain mostly the same information. We performed extensive ablation studies to confirm the effectiveness of the other parts of the model (Section 6.2 of our IJCAR'20 paper [7]), including the chopping heuristics and the attention and copy mechanisms. We also performed a generalization study which confirms that ROOSTERIZE can perform well on a new project with little additional training (Appendix D.3 of our IJCAR'20 paper [7]).

B. Qualitative

We carried out a qualitative case study using ROOSTERIZE by applying it to the FCSL PCM Coq project, which comprises 690 lemmas. 36 suggestions (5%) exactly matched the existing lemma names. We then asked the project maintainer to comment on the remaining suggestions. The maintainer found that 20% of the suggested names he inspected were of good quality, out of which more than half were of high quality. Considering that the analysis was of top-1 suggestions, we find these results encouraging.

VI. LIMITATIONS AND FUTURE WORK

Due to limitations in the protocol that VSCode uses to communicate with Coq, our VSCode extension cannot obtain name suggestions for a lemma in real time as it is being edited, i.e., by monitoring changes to the Coq source file and proof state. However, our toolchain can support this mode of use once protocol limitations are lifted.

The quality of lemma name suggestions is highly dependent on the quality of the pre-trained neural networks, and building a model requires careful curation of Coq training data. While we have constructed such a high-quality dataset based on the MathComp family of projects, additional datasets must be curated to suggest names that follow conventions other than those for MathComp.

VII. CONCLUSION

We presented ROOSTERIZE, a toolchain for suggesting lemma names in Coq verification projects. Nearly all related work addresses fundamentally different name generation tasks in conventional languages such as Java [1]. An exception is Aspinall and Kaliszyk [5], who learn naming from a corpus for the HOL Light proof assistant; however, their technique only suggests names that appear in the training data. ROOSTERIZE uses novel neural network models pre-trained on existing Coq code to generate lemma names. Our quantitative evaluation showed that ROOSTERIZE outperforms several strong baselines, and our qualitative evaluation demonstrated the quality of generated lemma names. We believe ROOSTERIZE can be especially useful to proof engineers in large Coq projects to ensure that lemma names follow prevailing conventions. Through our integration of ROOSTERIZE with the VSCode editor, naming suggestions can be continually provided as Coq code is added and revised.

ACKNOWLEDGMENTS

We thank Cyril Cohen, Emilio Jesús Gallego Arias, Anton Trunov, and the anonymous reviewers for their comments and feedback. This work was partially supported by the US National Science Foundation under Grant No. CCF-1652517 and the University of Texas at Austin Continuing Fellowship.

REFERENCES

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *FSE*, 2014.
[2] Coq Development Team, "The Coq proof assistant, version 8.10.0," Oct. 2019. [Online]. Available: https://zenodo.org/record/3476303
[3] X. Leroy, "Formal verification of a realistic compiler," *Commun. ACM*, vol. 52, no. 7, 2009.
[4] I. Sergey, J. R. Wilcox, and Z. Tatlock, "Programming and proving with distributed protocols," *PACMPL*, vol. 2, no. POPL, 2018.
[5] D. Aspinall and C. Kaliszyk, "What's in a theorem name?" in *ITP*, 2016.
[6] C. Cohen and A. Trunov, "Contribution guide for the Mathematical Components library," 2018, last accessed 2020-11-20. [Online]. Available: https://github.com/math-comp/math-comp/blob/mathcomp-1.9.0/CONTRIBUTING.md
[7] P. Nie, K. Palmskog, J. J. Li, and M. Gligoric, "Deep generation of Coq lemma names using elaborated terms," in *IJCAR*, 2020.
[8] E. J. Gallego Arias, "SerAPI: Machine-friendly, data-centric serialization for Coq," MINES ParisTech, Tech. Rep., 2016, https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408.
[9] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in PyTorch," in *Autodiff Workshop*, 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ
[10] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," in *ACL Demo*, 2017.
[11] Microsoft, "Visual Studio Code Website," 2020, last accessed 2020-11-20. [Online]. Available: https://code.visualstudio.com
[12] Microsoft, "Language Server Protocol Website," 2020, last accessed 2020-11-19. [Online]. Available: https://microsoft.github.io/language-server-protocol/
[13] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *NeurIPS*, 2014.
[14] T. Luong, H. Pham, and C. D. Manning, "Effective approaches to attention-based neural machine translation," in *EMNLP*, 2015.
[15] A. See, P. J. Liu, and C. D. Manning, "Get to the point: Summarization with pointer-generator networks," in *ACL*, 2017.
[16] OPAM Development Team, "OCaml Package Manager," 2020, last accessed 2020-11-19. [Online]. Available: https://opam.ocaml.org
[17] Anaconda, Inc., "Miniconda - Conda documentation," 2020, last accessed 2020-11-19. [Online]. Available: https://docs.conda.io/en/latest/miniconda.html
[18] Coq Development Team, "Building a Coq project," 2019, last accessed 2020-11-20. [Online]. Available: https://coq.inria.fr/distrib/V8.10.2/refman/practical-tools/utilities.html
[19] P. Nie, K. Palmskog, J. J. Li, and M. Gligoric, "MathComp Corpus of Coq Code," 2020, last accessed 2020-11-20. [Online]. Available: https://github.com/EngineeringSoftware/math-comp-corpus
[20] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *ACL*, 2002.