Deep Generation of Coq Lemma Names Using Elaborated Terms

Pengyu Nie[1(✉)], Karl Palmskog[2], Junyi Jessy Li[1], and Milos Gligoric[1]

[1] The University of Texas at Austin, Austin, TX, USA
{pynie,gligoric}@utexas.edu, jessy@austin.utexas.edu
[2] KTH Royal Institute of Technology, Stockholm, Sweden
palmskog@kth.se

Abstract. Coding conventions for naming, spacing, and other essentially stylistic properties are necessary for developers to effectively understand, review, and modify source code in large software projects. Consistent conventions in verification projects based on proof assistants, such as Coq, increase in importance as projects grow in size and scope. While conventions can be documented and enforced manually at high cost, emerging approaches automatically learn and suggest idiomatic names in Java-like languages by applying statistical language models on large code corpora. However, due to its powerful language extension facilities and fusion of type checking and computation, Coq is a challenging target for automated learning techniques. We present novel generation models for learning and suggesting lemma names for Coq projects. Our models, based on multi-input neural networks, are the first to leverage syntactic and semantic information from Coq's lexer (tokens in lemma statements), parser (syntax trees), and kernel (elaborated terms) for naming; the key insight is that learning from elaborated terms can substantially boost model performance. We implemented our models in a toolchain, dubbed Roosterize, and applied it on a large corpus of code derived from the Mathematical Components family of projects, known for its stringent coding conventions. Our results show that Roosterize substantially outperforms baselines for suggesting lemma names, highlighting the importance of using multi-input models and elaborated terms.

Keywords: Proof assistants · Coq · Lemma names · Neural networks

1 Introduction

Programming language source code with deficient coding conventions, such as misleading function and variable names and irregular spacing, is difficult for developers to effectively understand, review, and modify [8,52,67]. Code with haphazard adherence to conventions may also be more bug-prone [17]. The problem is exacerbated in large projects with many developers, where different source code files and components may have inconsistent and clashing conventions.

© Springer Nature Switzerland AG 2020

Many open source software projects manually document coding conventions that contributors are expected to follow, and maintainers willingly accept fixes of violations to such conventions [2]. Enforcement of conventions can be performed by static analysis tools [30,59]. However, such tools require developers to write precise checks for conventions, which are tedious to define and often *incomplete*. To address this problem, researchers have proposed techniques for automatically learning coding conventions for Java-like languages from code corpora by applying statistical language models [4]. These models are applicable because code in these languages has high *naturalness* [35], i.e., statistical regularities and repetitiveness. Learned conventions can then be used to, e.g., suggest names in code.

Proof assistants, such as Coq [15], are increasingly used to formalize results in advanced mathematics [28,29] and develop large trustworthy software systems, e.g., compilers, operating systems, file systems, and distributed systems [18,44,73]. Such projects typically involve contributions of many participants over several years, and require considerable effort to maintain over time. Coding conventions are essential for evolution of large verification projects, and are thus highly emphasized in the Coq libraries HoTT [37] and Iris [39], in Lean's Mathlib [9], and in particular in the influential Mathematical Components (MathComp) *family of Coq projects* [19]. Extensive changes to adhere to conventions, e.g., on naming, are regularly requested by MathComp maintainers for proposed external contributions [50], and its conventions have been adopted, to varying degrees, by a growing number of independent Coq projects [1,13,24,66].

We believe these properties make Coq code related to MathComp an attractive target for automated learning and suggesting of coding conventions, in particular, for suggesting *lemma names* [7]. However, serious challenges are posed by, on the one hand, Coq's powerful language extension facilities and fusion of type checking and computation [12], and on the other hand, the idiosyncratic conventions used by Coq practitioners compared to software engineers. Hence, although suggesting lemma names is similar in spirit to suggesting method names in Java-like languages [74], the former task is more challenging in that lemma names are typically much shorter than method names and tend to include heavily abbreviated terminology from logic and advanced mathematics; a single character can carry significant information about a lemma's meaning. For example, the MathComp lemma names `card_support_normedTI` ("cardinality of support groups of a normed trivial intersection group") and `extprod_mulgA` ("associativity of multiplication operations in external product groups") concisely convey information on lemma statement structure and meaning through both abbreviations and suffixes, as when the suffix `A` indicates an associative property.

In this paper, we present novel generation models for learning and suggesting lemma names for Coq verification projects that address these challenges. Specifically, based on our knowledge of Coq and its implementation, we developed multi-input encoder-decoder neural networks for generating names that use information directly from Coq's internal data structures related to lexing, parsing, and type checking. In the context of naming, our models are the first to leverage the *lemma lemma statement* as well as the corresponding *syntax tree* and *elaborated term* (which we call *kernel tree*) processed by Coq's kernel [53].

We implemented our models in a toolchain, dubbed Roosterize, which we used to learn from a high-quality Coq corpus derived from the MathComp family. We then measured the performance of Roosterize using automatic metrics, finding that it significantly outperforms baselines. Using our best model, we also suggested lemma names for the PCM library [56,66], which were manually reviewed by the project maintainer with encouraging results.

To allow Roosterize to use information directly from Coq's lexer, parser, and kernel, we extended the SerAPI library [26] to serialize Coq tokens, syntax trees, and kernel trees into a machine-readable format. This allowed us to achieve robustness against user-defined notations and other extensions to Coq syntax. Thanks to our integration with SerAPI and its use of metaprogramming, we expect our toolchain to only require modest maintenance as Coq evolves.
We make the following key contributions in this work:

- **Models**: We propose novel generation models based on multi-input neural networks to learn and suggest lemma names for Coq verification projects. A key property of our models is that they combine data from several Coq phases, including lexing, parsing, and term elaboration.
- **Corpus**: Advised by MathComp developers, we constructed a corpus of high-quality Coq code for learning coding conventions, totaling over 164k LOC taken from four core projects. We believe that our corpus can enable development of many novel techniques for Coq based on statistical language models.
- **Toolchain**: We implemented a toolchain, dubbed Roosterize, which suggests lemma names for a given Coq project. We envision Roosterize being useful during the review process of proposed contributions to a Coq project.
- **Evaluation**: We performed several experiments with Roosterize to evaluate our models using our corpus. Our results show that Roosterize performs significantly better than several strong baselines, as measured by standard automatic metrics [60]. The results also reveal that our novel multi-input models, as well as the incorporation of kernel trees, are important for suggestion quality. Finally, we performed a manual quality analysis by suggesting lemma names for a medium sized Coq project [56], evaluated by its maintainer, who found many of the suggestions useful for improving naming consistency.

The appendix of the extended version of the paper [57] describes more experiments, including an automatic evaluation on additional Coq projects. We provide artifacts related to our toolchain and corpus at: https://github.com/EngineeringSoftware/roosterize.

2 Background

This section gives brief background related to Coq and the Mathematical Components (MathComp) family of projects, as well as the SerAPI library.

```
1 Lemma mg_eq_proof L1 L2 (N1 : mgClassifier L1) : L1 =i L2 -> nerode L2 N1.
2 Proof. move => H0 u v. split => [/nerodeP H1 w|H1].
3   - by rewrite -!H0.
4   - apply/nerodeP => w. by rewrite !H0.
5 Qed.
```

Fig. 1. Coq lemma on the theory of regular languages, including proof script.

Coq and Gallina: Coq is a proof assistant based on dependent types, implemented in the OCaml language [15,20]. For our purposes, we view Coq as a programming language and type-checking toolchain. Specifically, Coq *files* are sequences of *sentences*, with each sentence ending with a period. Sentences are essentially either (a) commands for printing and other output, (b) definitions of functions, lemmas, and datatypes in the Gallina language [21], or (c) expressions in the Ltac tactic language [22]. We will refer to definitions of lemmas as in (b) as *lemma sentences*. Coq internally represents a lemma sentence both as a sequence of tokens (lexing phase) and as a syntax tree (parsing phase).

In the typical workflow for a Coq-based verification project, users write datatypes and functions and then interactively prove lemmas about them by executing different tactic expressions that may, e.g., discharge or split the current proof goal. Both statements to be proved and proofs are represented internally as *terms* produced during an *elaboration* phase [53]; we refer to elaborated terms as *kernel trees*. Hence, as tactics are successfully executed, they gradually build a kernel tree. The `Qed` command sends the kernel tree for a tentative proof to Coq's kernel for final certification. We call a collection of Ltac tactic sentences that build a kernel tree a *proof script*.

Figure 1 shows a Coq lemma and its proof script, taken verbatim from a development on the theory of regular languages [24]. Line 1 contains a lemma sentence with the lemma name `mg_eq_proof`, followed by a *lemma statement* (on the same line) involving the arbitrary languages `L1` and `L2`, i.e., typed variables that are implicitly universally quantified. When Coq processes line 5, the kernel certifies that the kernel tree generated by the proof script (lines 2 to 4) has the type (is a proof) of the kernel tree for the lemma statement on line 1.

MathComp and Lemma Naming: The MathComp family of Coq projects, including in particular the MathComp library of general mathematical definitions and results [49], grew out of Gonthier's proof of the four-color theorem [28], with substantial developments in the context of the landmark proof of the odd order theorem in abstract algebra [29]. The MathComp library is now used in many projects outside of the MathComp family, such as in the project containing the lemma in Fig. 1 [23]. MathComp has documented naming conventions for two kinds of entities: (1) variables and (2) functions and lemmas [19]. Variable names tend to be short and simple, while function and lemma names can be long and consist of several *name components*, typically separated by an underscore, but sometimes using CamelCase. Examples of definition and lemma names in Fig. 1 include `mg_eq_proof`, `mgClassifier`, `nerode`, and `nerodeP`. Note that lemma

```
Lemma mg_eq_proof L1 L2 (N1 : mgClassifier L1) : L1 =i L2 -> nerode L2 N1.      sentence
```

```
(Sentence((IDENT Lemma)(IDENT mg_eq_proof)(IDENT L1)(IDENT L2)                      tokens
  (KEYWORD"(")(IDENT N1)(KEYWORD :)(IDENT mgClassifier)
  (IDENT L1)(KEYWORD")")(KEYWORD :)(IDENT L1)(KEYWORD =i)(IDENT L2)
  (KEYWORD ->)(IDENT nerode)(IDENT L2)(IDENT N1)(KEYWORD .)))
```

```
(VernacExpr()(VernacStartTheoremProof Lemma (Id mg_eq_proof)                   syntax tree
 (((CLocalAssum(Name(Id L1))(CHole()IntroAnonymous()))
   (CLocalAssum(Name(Id L2))(CHole()IntroAnonymous()))
   (CLocalAssum(Name(Id N1))
    (CApp(CRef(Ser_Qualid(DirPath())(Id mgClassifier)))(CRef(Ser_Qualid(DirPath())(Id L1))))))
  (CNotation(InConstrEntrySomeLevel"_ -> _")
   (CNotation(InConstrEntrySomeLevel"_ =i _")
    (CRef(Ser_Qualid(DirPath())(Id L1)))(CRef(Ser_Qualid(DirPath())(Id L2))))
   (CApp(CRef(Ser_Qualid(DirPath())(Id nerode)))
    (CRef(Ser_Qualid(DirPath())(Id L2)))(CRef(Ser_Qualid(DirPath())(Id N1))))))))
```

```
(Prod (Name (Id char)) ... (Prod (Name (Id L1)) ...                           kernel tree
  (Prod (Name (Id L2)) ... (Prod (Name (Id N1)) ...
   (Prod Anonymous (App (Ref (DirPath ((Id ssrbool) (Id ssr) (Id Coq))) (Id eq_mem)) ...
     (Var (Id L1)) ... (Var (Id L2)))
    (App (Ref (DirPath ((Id myhill_nerode) (Id RegLang))) (Id nerode)) ...
     (Var (Id L2)) ... (Var (Id N1))))))))
```

Fig. 2. Coq lemma sentence at the top, with sexps for, from just below to bottom: tokens, syntax tree, and kernel tree; the lemma statement in each is highlighted.

names sometimes have *suffixes* to indicate their meaning, such as `P` in `nerodeP` which says that the lemma is a *characteristic property*. Coq functions tend to be named based on corresponding function definition bodies rather than just types (of the parameters and return value), analogously to methods in Java [47]. In contrast, MathComp lemma names tend to be based solely on the lemma statement. Hence, a more suitable name for the lemma in Fig. 1 is `mg_eq_nerode`.

SerAPI and Coq Serialization: SerAPI is an OCaml library and toolchain for machine interaction with Coq [26], which provides serialization and deserialization of Coq internal data structures to and from S-expressions (sexps) [51]. SerAPI is implemented using OCaml's PPX metaprogramming facilities [58], which enable modifying OCaml program syntax trees at compilation time. Figure 2 shows the lemma sentence on line 1 in Fig. 1, and below it, the corresponding (simplified) sexps for its tokens, syntax tree, and kernel tree, with the lemma statement highlighted in each representation. Note that the syntax tree omits the types of some quantified variables, e.g., for the types of `L1` and `L2`, as indicated by the `CHole` constructor. Note also that during elaboration of the syntax tree into the kernel tree by Coq, an implicit variable `char` is added (all-quantified via `Prod`), and the extensional equality operator `=i` is translated to its globally unique *kernel name*, `Coq.ssr.ssrbool.eq_mem`. Hence, a kernel tree can be much larger and contain more information than the corresponding syntax tree.

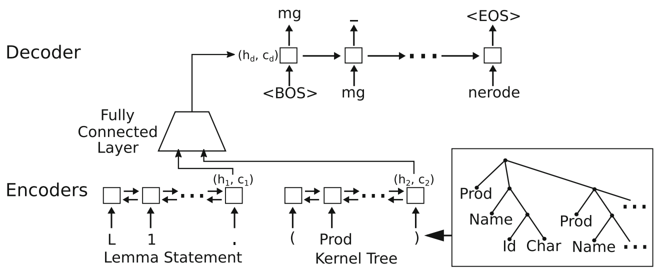

Fig. 3. Core architecture of our multi-input encoder-decoder models.

3 Models

In this section, we describe our multi-input generation models for suggesting Coq lemma names. Our models consider lemma name generation with an *encoder-decoder* mindset, i.e., we use neural architectures specifically designed for transduction tasks [68]. This family of architectures is commonly used for sequence generation, e.g., in machine translation [11] and code summarization [43], where it has been found to be much more effective than traditional probabilistic and retrieval-based approaches.

3.1 Core Architecture

Our 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 decoder—another RNN—generates the descriptive lemma name as a sequence. The model is trained end-to-end, maximizing the probability of the generated lemma name given the input. In contrast to prior work in language-code tasks that uses a single encoder [27], we design multi-input models that leverage both syntactic and semantic information from Coq's lexer, parser, and kernel. A high-level visualization of our architecture is shown in Fig. 3.

Encoding: Our multi-input encoders combine different kinds of syntactic and semantic information in the encoding phase. We use a different encoder for each input, which are: lemma statement, syntax tree, and kernel tree.

Coq data structure instances can be large, with syntax trees having an average depth of 28.03 and kernel trees 46.51 in our corpus (we provide detailed statistics in Sect. 4). Therefore, we flatten the trees into sequences, which can be trained more efficiently than tree encoders without performance loss [38]. We flatten the trees with pre-order traversal, and we use "(" and ")" as the boundaries of the children of a node. In later parts of this paper, we use syntax and kernel trees to refer to their flattened versions. In Sect. 3.2, we introduce *tree chopping* to reduce the length of the resulting sequences.

To encode lemma statements and flattened tree sequences, we use bi-directional Long-Short Term Memory (LSTM) [36] networks. LSTMs are advanced

```
(Prod Anonymous (App (Ref (DirPath ((Id ssrbool) (Id ssr) (Id Coq))) (Id eq_mem)) ...
  ((App (Ref ... ))) ... ))
```

```
(Prod Anonymous (App eq_mem ... (App (Ref ... )) ... ))
```

Fig. 4. Kernel tree sexp before and after chopping; chopped parts are highlighted.

RNNs good at capturing long-range dependencies in a sequence, and are widely used in encoders [38]. A bi-directional LSTM learns stronger representations (than a uni-directional LSTM) by encoding a sequence from both left to right and right to left [75].

Decoding: We use an LSTM (left to right direction only) as our decoder. To obtain the initial hidden and cell states (h_d, c_d) of the decoder, we learn a unified representation of these separate encoders by concatenating their final hidden and cell states (h_i, c_i), and then applying a fully connected layer on the concatenated states: $h_d = W_h \cdot \texttt{concat}([h_i]) + b_h$ and $c_d = W_c \cdot \texttt{concat}([c_i]) + b_c$, where W_h, W_c, b_h, and b_c are learnable parameters.

During training, we maximize the log likelihood of the reference lemma name given all input sequences. Standard beam search is used to reduce the search space for the optimal sequence of tokens. With regular decoding, at each time step the decoder generates a new token relying on the preceding *generated* token, which can be error-prone and leads to slow convergence and instability. We mitigate this problem by performing decoding with teacher forcing [72] such that the decoder relies on the preceding *reference* token. At test time, the decoder still uses the proceeding generated token as input.

Attention: With RNN encoders, the input sequence is compressed into the RNN's final hidden states, which results in a loss of information, especially for longer sequences. The attention mechanism [48] grants the decoder access to the encoder hidden and cell states for all previous tokens. At each decoder time step, an attention vector is calculated as a distribution over all encoded tokens, indicating which token the decoder should "pay attention to". To make the attention mechanism work with multiple encoders, we concatenate the hidden states of the n encoders $[h_1, ..., h_n]$ and apply an attention layer on the result [70].

Initialization: Since there are no pre-trained token embeddings for Coq, we initialize each unique token in the vocabulary with a random vector sampled from the uniform distribution $U(-0.1, 0.1)$. These embeddings are trained together with the model. The hidden layer parameters of the encoders and decoders are also initialized with random vectors sampled from the same uniform distribution.

3.2 Tree Chopping

While syntax and kernel trees for lemma statements can be large, not all parts of the trees are relevant for naming. For instance, each constant reference is expanded to its fully qualified form in the kernel tree, but the added prefixes are usually related to directory paths and likely do not contain relevant information for generating the name of the lemma. Irrelevant information in long sequences can be detrimental to the model, since the model would have to reason about and encode all tokens in the sequence.

To this end, we implemented *chopping* heuristics for both syntax trees and kernel trees to remove irrelevant parts. The heuristics essentially: (1) replace the fully qualified name sub-trees with only the last component of the name; (2) remove the location information from sub-trees; (3) extract the singletons, i.e., non-leaf nodes that have only one child. Figure 4 illustrates the chopping of a kernel tree, with the upper box showing the tree before chopping with the parts to be removed highlighted, and the lower box showing the tree after chopping. In the example in the figure, we chopped a fully qualified name and extracted a singleton. These heuristics greatly reduce the size of the tree: for kernel trees, they reduce the average depth from 39.20 to 11.39.

Our models use chopped trees as the inputs to the encoders. As we discuss in more detail in Sect. 6, the chopped trees help the models to focus better on the relevant parts of the inputs. While the attention mechanism in principle could learn what the relevant parts of the trees are, our evaluation shows that it can easily be overwhelmed by large amounts of irrelevant information.

3.3 Copy Mechanism

We found it common for lemma name tokens to only occur in a single Coq file, whence they are unlikely to appear in the vocabulary learned from the training set, but can still appear in the respective lemma statement, syntax tree, or kernel tree. For example, `mg` occurs in both the lemma name and lemma statement in Fig. 1, but not outside the file the lemma is in. To account for this, we adopt the copy mechanism [64] which improves the generalizability of our model by allowing the decoder to *copy* from inputs rather than always choosing one word from the fixed vocabulary from the training set. To handle multiple encoders, similar to what we did with the attention layer, we concatenate the hidden states of each encoder and apply a copy layer on the concatenated hidden states.

3.4 Sub-tokenization

We sub-tokenize all inputs (lemma statements, syntax and kernel trees) and outputs (lemma names) in a pre-processing step. Previous work on learning from software projects has shown that sub-tokenization helps to reduce the sparsity of the vocabulary and improves the performance of the model [10]. However, unlike Java-like languages where the method names (almost) always follow the CamelCase convention, lemma names in Coq use a mix of snake_case, CamelCase, prefixes, and suffixes, thus making sub-tokenization more complex. For example, `extprod_mulgA` should be sub-tokenized to `extprod`, `_`, `mul`, `g`, and `A`.

To perform sub-tokenization, we implemented a set of heuristics based on the conventions outlined by MathComp developers [19]. After sub-tokenization, the vocabulary size of lemma names in our corpus was reduced from 8,861 to

Table 1. Projects from the MathComp family used in our corpus.

Project		SHA	#Files	#Lemmas	#Toks	LOC		LOC/file	
						Spec.	Proof	Spec.	Proof
finmap		27642a8	4	940	78,449	4,260	2,191	1,065.00	547.75
fourcolor		0851d49	60	1,157	560,682	9,175	27,963	152.92	466.05
math-comp		748d716	89	8,802	1,076,096	38,243	46,470	429.70	522.13
odd-order		ca602a4	34	367	519,855	11,882	24,243	349.47	713.03
Avg.		N/A	46.75	2,816.50	558,770.50	15,890.00	25,216.75	339.89	539.40
Σ		N/A	187	11,266	2,235,082	63,560	100,867	63,560	100,867

Table 2. Statistics on the lemmas in the training, validation, and testing sets.

	#Files	#Lemmas	Name		Stmt	
			#Char	#SubToks	#Char	#SubToks
training	152	8,861	10.14	4.22	44.16	19.59
validation	18	1,085	9.20	4.20	38.28	17.30
testing	17	1,320	9.76	4.34	48.49	23.20

2,328. When applying the sub-tokenizer on the lemma statements and syntax and kernel trees, we sub-tokenize the identifiers and not the keywords or operators.

3.5 Repetition Prevention

We observed that decoders often generated repeated tokens, e.g., `mem_mem_mem`. This issue also exists in natural language summarization [69]. We further observed that it is very unlikely to have repeated sub-tokens in lemma names used by proof engineers (only 1.37% of cases in our corpus). Hence, we simply forbid the decoder from repeating a sub-token (modulo "_") during beam search.

4 Corpus

We constructed a corpus of four large Coq projects from the MathComp family, totaling 164k lines of code (LOC). We selected these projects based on the recommendation of MathComp developers, who emphasized their high quality and stringent adherence to coding conventions. Our corpus is *self-contained*: there are inter-project dependencies within the corpus, but no project depends on a project outside the corpus (except Coq's standard library). All projects build with Coq version 8.10.2. Note that we need to be able to build projects to be able to extract tokens, syntax trees, and kernel trees.

Constituent Projects: Table 1 lists the projects in the corpus, along with basic information about each project. The table includes columns for the project identifier, revision SHA, number of files (#Files), number of lemmas (#Lemmas), number of tokens (#Toks), LOC for specifications (Spec.) and proof scripts

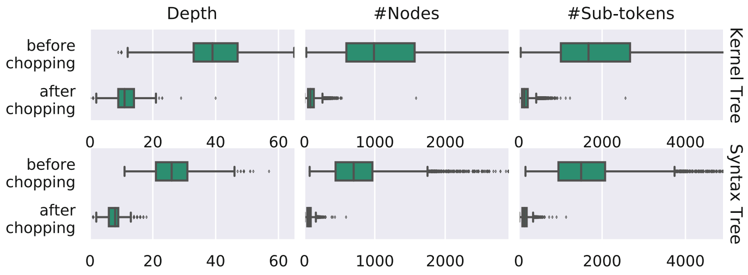

Fig. 5. Statistics on syntax and kernel trees.

(Proof), and average LOC per file for specifications and proof scripts. The math-comp SHA corresponds to version 1.9.0 of the library. The LOC numbers are computed with Coq's bundled `coqwc` tool. The last two rows of the table show the averages and sums across all projects.

Corpus Statistics: We extracted all lemmas from the corpus, and initially we obtained 15,005 lemmas in total. However, we found several outlier lemmas where the lemma statement, syntax tree and kernel tree were very large. To ensure stable training, and similar to prior work on generating method names for Java [47], we excluded the lemmas with the deepest 25% kernel trees. This left us with 11,266 lemmas. Column 4 of Table 1 shows the number of lemmas after filtering.

We randomly split corpus files into training, validation, and testing sets which contain 80%, 10%, 10% of the files, respectively. Table 2 shows statistics on the lemmas in each set, which includes columns for the number of files, the number of lemmas, the average number of characters and sub-tokens in lemma names, and the average number of characters and sub-tokens in lemma statements.

Figure 5 illustrates the changes of the depth, number of nodes and number of sub-tokens (after flattening) of the kernel trees (first row) and syntax trees (second row) before and after chopping. Our chopping process reduced tree depth by 70.9% for kernel trees and 70.7% for syntax trees, and reduced the number of nodes by 91.5% for kernel trees and 90.8% for syntax trees; after flattening, the resulting average sequence length is, for kernel trees 165 comparing to the original 2,056, and for syntax trees 144 comparing to the original 1,590. We provide additional statistics on lemmas before filtering in the appendix of the extended paper [57].

5 Implementation

In this section, we briefly describe our toolchain which implements the models in Sect. 3 and processes and learns from the corpus in Sect. 4; we dub this toolchain Roosterize. The components of the toolchain can be divided into two categories: (1) components that interact with Coq or directly process information extracted from Coq, and (2) components concerned with machine learning and name generation.

The first category includes several OCaml-based tools integrated with SerAPI [26] (and thus Coq itself), and Python-based tools for processing of data obtained via SerAPI from Coq. All OCaml tools have either already been included in, or accepted for inclusion into, SerAPI itself. The tools are as follows:

`sercomp`: We integrated the existing program `sercomp` distributed with SerAPI into Roosterize to serialize Coq files to lists of sexps for syntax trees.

`sertok`: We developed an OCaml program dubbed `sertok` on top of SerAPI. The program takes a Coq file as input and produces sexps of all tokens found by Coq's lexer in the file, organized at the sentence level.

`sername`: We developed an OCaml program dubbed `sername` on top of SerAPI. The program takes a list of fully qualified (kernel) lemma names and produces sexps for the kernel trees of the corresponding lemma statements.

`postproc` & `subtokenizer`: We created two small independent tools in Python to post-process Coq sexps and perform sub-tokenization, respectively.

For the second category, we implemented our machine learning models in Python using two widely-used deep learning libraries: PyTorch [61] and OpenNMT [41]. More specifically, we extended the sequence-to-sequence models in OpenNMT to use multi-input encoders, and extended attention and copy layers to use multiple inputs. Source code for the components of Roosterize is available from: https://github.com/EngineeringSoftware/roosterize.

6 Evaluation

This section presents an extensive evaluation of our models as implemented in Roosterize. Our automatic evaluation (Sect. 6.2) compares Roosterize with a series of strong baselines and reports on ablation experiments; additional experiments, e.g., on chopping heuristics, are described in the appendix of the extended version of the paper [57]. Our manual quality assessment (Sect. 6.3) analyzes 150 comments we received from the maintainer of the PCM library on names suggested by Roosterize for that project using our best model.

6.1 Models and Baselines

We study the combinations of: (1) using individual input (lemma statement and trees) in a single encoder, or multi-input encoders with different mixture of these inputs; and (2) using the attention and copy mechanisms. Our inputs include: lemma statement (*Stmt*), syntax tree (*SynTree*), chopped syntax tree (*ChopSynTree*), kernel tree (*KnlTree*), and chopped kernel tree (*ChopKnlTree*). For multiple inputs, the models are named by concatenating inputs with "+"; a "+" is also used to denote the presence of attention (*attn*) or copy (*copy*). For example, Stmt+ChopKnlTree+attn+copy refers to a model that uses two encoders—one for lemma statement and one for chopped kernel tree—and uses attention and copy mechanisms.

Table 3. Results of Roosterize models.

Group	Model	BLEU	Frag.Acc.	Top1	Top5
Multi-input +attn +copy	Stmt+ChopKnlTree+ChopSynTree+attn+copy	45.4	22.2%	7.5%	16.5%
	Stmt+ChopKnlTree+attn+copy	**47.2**	**24.9%**	**9.6%**	**18.0%**
	Stmt+ChopSynTree+attn+copy	37.7	18.1%	6.1%	10.6%
	ChopKnlTree+ChopSynTree+attn+copy	45.4	22.9%	7.6%	15.3%
Single-input +attn +copy	ChopKnlTree+attn+copy	42.9	19.8%	5.0%	11.7%
	ChopSynTree+attn+copy	39.8	18.3%	6.8%	12.2%
	KnlTree+attn+copy	37.0	14.2%	2.2%	8.4%
	SynTree+attn+copy	31.0	10.8%	2.8%	6.1%
	Stmt+attn+copy	38.9	19.4%	6.9%	11.6%
Multi-input +attn	Stmt+ChopKnlTree+ChopSynTree+attn	24.5	8.6%	0.4%	0.9%
	Stmt+ChopKnlTree+attn	25.6	8.5%	0.9%	1.7%
	Stmt+ChopSynTree+attn	23.8	8.2%	0.8%	1.6%
	ChopKnlTree+ChopSynTree+attn	28.4	10.9%	1.8%	3.4%
Single-input +attn	ChopKnlTree+attn	19.5	4.9%	0.6%	1.3%
	ChopSynTree+attn	28.9	12.1%	1.5%	2.9%
	KnlTree+attn	14.1	1.6%	0.0%	0.0%
	SynTree+attn	8.8	1.0%	0.0%	0.0%
	Stmt+attn	26.9	11.1%	1.1%	2.5%
Multi-input	Stmt+ChopKnlTree+ChopSynTree	17.7	3.5%	0.1%	0.2%
	Stmt+ChopKnlTree	19.5	4.5%	0.1%	0.3%
	Stmt+ChopSynTree	12.6	0.6%	0.0%	0.0%
	ChopKnlTree+ChopSynTree	16.7	2.4%	0.0%	0.1%
Single-input	ChopKnlTree	15.5	1.6%	0.0%	0.0%
	ChopSynTree	14.5	0.8%	0.1%	0.1%
	KnlTree	12.0	0.6%	0.0%	0.0%
	SynTree	5.7	0.4%	0.0%	0.0%
	Stmt	20.0	4.7%	0.1%	0.3%
-	Retrieval-based	28.3	10.0%	0.2%	0.3%

We consider the vanilla encoder-decoder models with only one input (lemma statement, kernel tree, or syntax tree) as baseline models. We also compare with a retrieval-based baseline model implemented using Lucene [6]: a k-nearest neighbors classifier using the tf-idf of the tokens in lemma statement as features.

Hyperparameters are tuned on the validation set within the following options: embedding dimensions from {200, 500, 1000}, number of hidden units in each LSTM from {200, 500, 1000}, number of stacked LSTM layers from {1, 2, 3}. We set the dropout rate between LSTM layers to 0.5. We set the output dimension of the fully connected layer for combining encoders to the same number as the number of hidden units in each LSTM. We checked the validation loss every 200 training steps (as defined in OpenNMT [41], which is similar to one training epoch on our dataset), and set an early stopping threshold of 3. We used the Adam [40] optimizer with a learning rate of 0.001. We used a beam size of 5 in beam search. All the experiments were run with one NVIDIA 1080-TI GPU and Intel Xeon E5-2620 v4 CPU.

6.2 Automatic Evaluation

Metrics: We use four automatic metrics which evaluate generated lemma names against the reference lemma name (as written by developers) in the testing set. Each metric captures a different level of granularity of the generation quality. *BLEU* [60] is a standard metric used in transduction tasks including language $\leftrightarrow$ code transduction. It calculates the number of n-grams in a generated sequence that also appear in the reference sequence, where one "n-gram" is n consecutive items in a sequence (in our case, one "n-gram" is n consecutive characters in the sequence of characters of the lemma name). We use it to compute the $1 \sim 4$-grams overlap between the characters in generated name and characters in the reference name, averaged between $1 \sim 4$-grams with smoothing method proposed by Lin and Och [46]. *Fragment accuracy* computes the accuracy of generated names on the fragment level, which is defined by splitting the name by underscores ("_"). For example, `map_determinant_mx` has a fragment accuracy of 66.7% when evaluated against `det_map_mx`. Unlike BLEU, fragment accuracy ignores the ordering of the fragments. Finally, *top-1 accuracy* and *top-5 accuracy* compute how often the true name fully matches the generated name or is one of the top-5 generated names.

Results: Table 3 shows the performance of the models. Similar models are grouped together. The first column shows the names of the model groups and the second column shows the names of the models. For each model, we show values for the four automatic metrics, BLEU, fragment accuracy (Frag.Acc.), top-1 accuracy (Top1), and top-5 accuracy (Top5). We repeated each experiment 3 times, with different random initialization each time, and computed the averages of each automated metric. We performed statistical significance tests—under significance level $p < 0.05$ using the bootstrap method [14]—to compare each pair of models. We use bold text to highlight the best value for each automatic metric, and gray background for baseline models. We make several observations:

Finding #1: The best overall performance (BLEU = 47.2) is obtained using the multi-input model with lemma statement and chopped kernel tree as inputs, which also includes copy and attention mechanisms (Stmt+ChopKnlTree+attn+copy). The improvements over all other models are statistically significant and all automatic metrics are consistent in identifying the best model. This shows the importance of using Coq's internal structures and focusing only on certain parts of those structures.

Finding #2: The copy mechanism brings statistically significant improvements to all models. This can be clearly observed by comparing groups 1 and 3 in the table, as well as groups 2 and 4. For example, BLEU for Stmt+attn and Stmt+attn+copy are 26.9 and 38.9, respectively. We believe that the copy mechanism plays an important role because many sub-tokens are specific to the file context and do not appear in the fixed vocabulary learned on the files in training set.

Finding #3: Using chopped trees greatly improves performance of models and the improvements brought by upgrading KnlTree to ChopKnlTree or SynTree to

Table 4. Manual quality analysis representative examples.

Lemma statement	Hand-written	Roosterize	Comment
`p s : supp (kfilter p s) = filter p (supp s)`	`supp_kfilt`	`supp_kfilter`	✓ Using only `kfilt` has cognitive overhead
`g e k v f : path ord k (supp f) -> foldfmap g e (ins k v f) = g (k, v) (foldfmap g e f)`	`foldf_ins`	`foldfmap_ins`	✓ The whole function name is used in the suggested name
`: transitive (@ord T)`	`trans`	`ord_trans`	✓ Useful to add the `ord` prefix to the name
`s : sorted (@ord T) s -> sorted (@oleq T) s`	`sorted_oleq`	`ord_sorted`	× The conclusion content should have greater priority
`x y : total_spec x y (ord x y) (x == y) (ord y x)`	`totalP`	`ordP`	× Maybe this lemma should be named `ord_totalP`?
`p1 p2 s : kfilter (predI p1 p2) s = kfilter p1 (kfilter p2 s)`	`kfilter_predI`	`eq_kfilter`	× The suggested name is too generic

ChopSynTree are statistically significant. For example, this can be clearly seen in the second group: BLEU for KnlTree+attn+copy and ChopKnlTree+attn+copy are 37.0 and 42.9, respectively. We believe that the size of the original trees, and a lot of irrelevant data in those trees, hurt the performance. The fact that ChopKnlTree and ChopSynTree both perform much better than using KnlTree or SynTree across all groups indicate that the chopped trees could be viewed as a form of supervised attention with flat values that helps later attention layers to focus better.

Finding #4: Although chopped syntax tree with attention outperforms (statistically significant) chopped kernel tree with attention (BLEU 28.9 vs. 19.5), chopped kernel tree with attention and copy by far outperforms (statistically significant) chopped syntax tree with attention and copy (BLEU 42.9 vs. 39.8). The copy mechanism helps kernel trees much more than the syntax trees, because the mathematical notations and symbols in the syntax trees get expanded to their names in the kernel trees, and some of them are needed as a part of the lemma names.

Finding #5: Lemma statement and syntax tree do not work well together, primarily because the two representations contain mostly the same information.

In which case, a model taking both as inputs may not work as well as using only one of the inputs, because more parameters need to be trained.

Finding #6: The retrieval-based baseline, which is the strongest among baselines, outperforms several encoder-decoder models without attention and copy or with only attention, but is worse than (statistically significant) all models with both attention and copy mechanisms enabled.

6.3 Manual Quality Analysis

While generated lemma names may not always match the manually written ones in the training set, they can still be semantically valid and conform to prevailing conventions. However, such name properties are not reflected in our automatic evaluation metrics, since these metrics only consider exactly matched tokens as correct. To obtain a more complete evaluation, we therefore performed a manual quality analysis of generated lemma names from Roosterize by applying it to a Coq project outside of our corpus, the PCM library [56]. This project depends on MathComp, and follows, to a degree, many of the MathComp coding conventions. The PCM library consists of 12 Coq files, and contains 690 lemmas.

We ran Roosterize with the best model (Stmt+ChopKnlTree+attn+copy) on the PCM library to get the top-1 suggestions for all lemma names. Overall, the Roosterize suggestions achieved a BLEU score of 36.3 and a fragment accuracy of 17%, and 36 suggestions (5%) exactly match the existing lemma names. Next, we asked the maintainer of the PCM library to evaluate the remaining 654 lemma names (those that do not match exactly) and send us feedback.

The maintainer spent one day on the task and provided comments on 150 suggested names. We analyzed these comments to identify patterns and trends. He 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 excluding exact matches, we find these figures encouraging. For low-quality names, a clear trend was that they were often "too generic". Similar observations have been made about the results from encoder-decoder models in dialog generation [45,65]. In contrast, useful suggestions were typically able to expand or elaborate on name components that are intuitively too concise, e.g., replacing `kfilt` with `kfilter`. Table 4 lists examples that are representative of these trends; checkmarks indicate useful suggestions, while crosses indicate unsuitability. We also include comments from the maintainer. As illustrated by the comments, even suggestions considered unsuitable may contain useful parts.

7 Discussion

Our toolchain builds on Coq 8.10.2, and thus we only used projects that support this version. However, we do not expect any fundamental obstacles in supporting future Coq releases. Thanks to the use of OCaml metaprogramming via PPX, which allowed eliding explicit references to the internal structure of Coq datatypes, SerAPI itself and our extensions to it are expected to require only modest effort to maintain as Coq evolves.

Our models and toolchain may not be applicable to Coq projects unrelated to the MathComp family of projects, i.e., projects which do not follow any MathComp conventions. To the best of our knowledge, MathComp's coding conventions are the most recognizable and well-documented in the Coq community; suggesting coding conventions based on learning from projects unrelated to MathComp are likely to give more ambiguous results that are difficult to validate manually. Our case study also included generating suggestions for a project outside the MathComp family, the PCM library, with encouraging results.

Our models are in principle applicable to proof assistants with similar foundations, such as Lean [54]. However, the current version of Lean, Lean 3, does not provide serialization of internal data structures as SerAPI does for Coq, which prevents direct application of our toolchain. Application of our models to proof assistants with different foundations and proof-checking toolchains, such as Isabelle/HOL, is even less straightforward, although the Archive of Formal Proofs (AFP) contains many projects with high-quality lemma names [25].

8 Related Work

Naturalness and Coding Conventions: Hindle et al. [35] first applied the concept of naturalness to Java-like languages, noting that program statement regularities and repetitiveness make statistical language models applicable for performing software engineering tasks [4]. Rahman et al. [62] validated the naturalness of other similar programming languages, and Hellendoorn et al. [31] found high naturalness in Coq code, providing motivation for our application of statistical language models to Coq. Allamanis et al. [2] used the concept of naturalness and statistical language models to learn and suggest coding conventions, including names, for Java, and Raychev et al. [63] used conditional random fields to learn and suggest coding conventions for JavaScript. To our knowledge, no previous work has developed *applications* of naturalness for proof assistants; Hellendorn et al. [31] only measured naturalness for their Coq corpus.

Suggesting Names: Prior work on suggesting names mostly concerns Java method names. Liu et al. [47] used a similarity matching algorithm, based on deep representations of Java method names and bodies learned with Paragraph Vector and convolutional neural networks, to detect and fix inconsistent Java method names. Allamanis et al. [3] used logbilinear neural language models supplemented by additional manual features to predict Java method and class names. Java method names have also been treated as short, descriptive "summaries" of its body; in this view, prior work has augmented attention mechanisms in convolutional networks [5], used sequence-to-sequence models to learn from descriptions (e.g., Javadoc comments) [27], and utilized the tree-structure of the code in a hierarchical attention network [74]. Unlike Java syntax trees, Coq syntax and kernel trees contain considerable semantic information useful for naming. In the work closest to our domain, Aspinall and Kaliszyk used a k-nearest neighbors multi-label classifier on a corpus for the HOL Light proof assistant to suggest names of lemmas [7]. However, their technique only suggests names that exist in the training data and therefore does not generalize. To our knowledge, ours is the first neural generation model for suggesting names in a proof assistant context.

Mining and Learning for Proof Assistants: Müller et al. [55] exported Coq kernel trees as XML strings to translate 49 Coq projects to the OMDoc theory graph format. Rather than translating documents to an independently specified format, we produce lightweight machine-readable representations of Coq's internal data structures. Wiedijk [71] collected early basic statistics on the core libraries of several proof assistants, including Coq and Isabelle/HOL. Blanchette et al. [16] mined the AFP to gather statistics such as the average number of lines of Isabelle/HOL specifications and proof scripts. However, these corpora were not used to perform learning. Komendantskaya et al. [32–34,42] used machine learning without neural networks to identify patterns in Coq tactic sequences and proof kernel trees, e.g., to find structural similarities between lemmas and simplify proof development. In contrast, our models capture similarity among several different representations of lemma *statements* to generate lemma names.

9 Conclusion

We presented novel techniques, based on neural networks, for learning and suggesting lemma names in Coq verification projects. We designed and implemented multi-input encoder-decoder models that use Coq's internal data structures, including (chopped) syntax trees and kernel trees. Additionally, we constructed a large corpus of high quality Coq code that will enable development and evaluation of future techniques for Coq. We performed an extensive evaluation of our models using the corpus. Our results show that the multi-input models, which use internal data structures, substantially outperform several baselines; the model that uses the lemma statement tokens and the chopped kernel tree with attention and copy mechanism performs the best. Based on our findings, we believe that multi-input models leveraging key parts of internal data structures can play a critical role in producing high-quality lemma name suggestions.

Acknowledgments. The authors thank Yves Bertot, Cyril Cohen, Emilio Jesús Gallego Arias, Gaëtan Gilbert, Hugo Herbelin, Anton Trunov, Théo Zimmermann, and the anonymous reviewers for their comments and feedback. This work was partially supported by the US National Science Foundation under Grant Nos. CCF-1652517 and IIS-1850153, and by the Swedish Foundation for Strategic Research under the TrustFull project.

References

1. Affeldt, R., Garrigue, J.: Formalization of error-correcting codes: from Hamming to modern coding theory. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 17–33. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-22102-1_2

2. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Learning natural coding conventions. In: International Symposium on the Foundations of Software Engineering, pp. 281–293. ACM, New York (2014). https://doi.org/10.1145/2635868.2635883
3. Allamanis, M., Barr, E.T., Bird, C., Sutton, C.: Suggesting accurate method and class names. In: Joint Meeting on Foundations of Software Engineering, pp. 38–49. ACM, New York (2015). https://doi.org/10.1145/2786805.2786849
4. Allamanis, M., Barr, E.T., Devanbu, P., Sutton, C.: A survey of machine learning for big code and naturalness. ACM Comput. Surv. **51**(4), 81:3–81:37 (2018). https://doi.org/10.1145/3212695
5. Allamanis, M., Peng, H., Sutton, C.: A convolutional attention network for extreme summarization of source code. In: International Conference on Machine Learning, pp. 2091–2100 (2016)
6. Apache Software Foundation: Apache Lucene (2020). https://lucene.apache.org. Accessed 23 Jan 2020
7. Aspinall, D., Kaliszyk, C.: What's in a theorem name? In: Blanchette, J.C., Merz, S. (eds.) ITP 2016. LNCS, vol. 9807, pp. 459–465. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-43144-4_28
8. Avidan, E., Feitelson, D.G.: Effects of variable names on comprehension: an empirical study. In: International Conference on Program Comprehension, pp. 55–65. IEEE Computer Society, Washington (2017). https://doi.org/10.1109/ICPC.2017.27
9. Avigad, J.: Mathlib naming conventions (2016). https://github.com/leanprover-community/mathlib/blob/snapshot-2019-10/docs/contribute/naming.md. Accessed 23 Jan 2020
10. Babii, H., Janes, A., Robbes, R.: Modeling vocabulary for big code machine learning. CoRR abs/1904.01873 (2019). https://arxiv.org/abs/1904.01873
11. Bahdanau, D., Cho, K., Bengio, Y.: Neural machine translation by jointly learning to align and translate. In: International Conference on Learning Representations (2015). https://arxiv.org/abs/1409.0473
12. Barendregt, H., Barendsen, E.: Autarkic computations in formal proofs. J. Autom. Reason. **28**(3), 321–336 (2002). https://doi.org/10.1023/A:1015761529444
13. Bartzia, E.-I., Strub, P.-Y.: A formal library for elliptic curves in the Coq proof assistant. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 77–92. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08970-6_6
14. Berg-Kirkpatrick, T., Burkett, D., Klein, D.: An empirical investigation of statistical significance in NLP. In: Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pp. 995–1005. Association for Computational Linguistics, Stroudsburg (2012)
15. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-662-07964-5
16. Blanchette, J.C., Haslbeck, M., Matichuk, D., Nipkow, T.: Mining the archive of formal proofs. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) CICM 2015. LNCS (LNAI), vol. 9150, pp. 3–17. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_1
17. Boogerd, C., Moonen, L.: Evaluating the relation between coding standard violations and faults within and across software versions. In: International Working Conference on Mining Software Repositories, pp. 41–50. IEEE Computer Society, Washington (2009). https://doi.org/10.1109/MSR.2009.5069479

18. Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M.F., Zeldovich, N.: Using crash Hoare logic for certifying the FSCQ file system. In: Symposium on Operating Systems Principles, pp. 18–37. ACM, New York (2015). https://doi.org/10.1145/2815400.2815402
19. Cohen, C.: Contribution guide for the Mathematical Components library (2018). https://github.com/math-comp/math-comp/blob/mathcomp-1.9.0/CONTRIBUTING.md. Accessed 14 Apr 2020
20. Coq Development Team: The Coq proof assistant, version 8.10.0, October 2019. https://doi.org/10.5281/zenodo.3476303
21. Coq Development Team: The Gallina specification language (2019). https://coq.inria.fr/distrib/V8.10.2/refman/language/gallina-specification-language.html. Accessed 17 Apr 2020
22. Delahaye, D.: A tactic language for the system Coq. In: Parigot, M., Voronkov, A. (eds.) LPAR 2000. LNAI, vol. 1955, pp. 85–95. Springer, Heidelberg (2000). https://doi.org/10.1007/3-540-44404-1_7
23. Doczkal, C., Kaiser, J.O., Smolka, G.: Regular language representations in Coq (2020). https://github.com/coq-community/reglang. Accessed 09 Apr 2020
24. Doczkal, C., Smolka, G.: Regular language representations in the constructive type theory of Coq. J. Autom. Reason. **61**(1), 521–553 (2018)
25. Eberl, M., Klein, G., Nipkow, T., Paulson, L., Thiemann, R.: Archive of Formal Proofs (2020). https://www.isa-afp.org. Accessed 23 Jan 2020
26. Gallego Arias, E.J.: SerAPI: machine-friendly, data-centric serialization for Coq. Technical report, MINES ParisTech (2016). https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408
27. Gao, S., Chen, C., Xing, Z., Ma, Y., Song, W., Lin, S.: A neural model for method name generation from functional description. In: International Conference on Software Analysis, Evolution and Reengineering, pp. 414–421. IEEE Computer Society, Washington (2019). https://doi.org/10.1109/SANER.2019.8667994
28. Gonthier, G.: Formal proof–the four-color theorem. Not. Am. Math. Soc. **55**(11), 1382–1393 (2008)
29. Gonthier, G., et al.: A machine-checked proof of the odd order theorem. In: Blazy, S., Paulin-Mohring, C., Pichardie, D. (eds.) ITP 2013. LNCS, vol. 7998, pp. 163–179. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39634-2_14
30. Google: Google-Java-Format (2020). https://github.com/google/google-java-format. Accessed 23 Jan 2020
31. Hellendoorn, V.J., Devanbu, P.T., Alipour, M.A.: On the naturalness of proofs. In: International Symposium on the Foundations of Software Engineering, New Ideas and Emerging Results, pp. 724–728. ACM, New York (2018). https://doi.org/10.1145/3236024.3264832
32. Heras, J., Komendantskaya, E.: ML4PG in computer algebra verification. In: Carette, J., Aspinall, D., Lange, C., Sojka, P., Windsteiger, W. (eds.) CICM 2013. LNCS (LNAI), vol. 7961, pp. 354–358. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39320-4_28
33. Heras, J., Komendantskaya, E.: Proof pattern search in Coq/SSReflect. CoRR abs/1402.0081 (2014). https://arxiv.org/abs/1402.0081
34. Heras, J., Komendantskaya, E.: Recycling proof patterns in Coq: case studies. Math. Comput. Sci. **8**(1), 99–116 (2014). https://doi.org/10.1007/s11786-014-0173-1

35. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: International Conference on Software Engineering, pp. 837–847. IEEE Computer Society, Washington (2012). https://doi.org/10.1109/ICSE.2012.6227135
36. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997). https://doi.org/10.1162/neco.1997.9.8.1735
37. HoTT authors: HoTT Conventions and Style Guide (2019). https://github.com/HoTT/HoTT/blob/V8.10/STYLE.md. Accessed 23 Jan 2020
38. Hu, X., Li, G., Xia, X., Lo, D., Jin, Z.: Deep code comment generation. In: International Conference on Program Comprehension, pp. 200–210. ACM, New York (2018). https://doi.org/10.1145/3196321.3196334
39. Iris authors: Iris Style Guide (2019). https://gitlab.mpi-sws.org/iris/iris/blob/iris-3.2.0/StyleGuide.md. Accessed 17 Apr 2020
40. Kingma, D.P., Ba, J.: Adam: a method for stochastic optimization. In: International Conference on Learning Representations (2015). https://arxiv.org/abs/1412.6980
41. Klein, G., Kim, Y., Deng, Y., Senellart, J., Rush, A.M.: OpenNMT: open-source toolkit for neural machine translation. In: Annual Meeting of the Association for Computational Linguistics, System Demonstrations, pp. 67–72. Association for Computational Linguistics, Stroudsburg (2017). https://doi.org/10.18653/v1/P17-4012
42. Komendantskaya, E., Heras, J., Grov, G.: Machine learning in Proof General: interfacing interfaces. In: Kaliszyk, C., Lüth, C. (eds.) International Workshop on User Interfaces for Theorem Provers. EPTCS, vol. 118, pp. 15–41. Open Publishing Association, Sydney (2013). https://doi.org/10.4204/EPTCS.118.2
43. LeClair, A., Jiang, S., McMillan, C.: A neural model for generating natural language summaries of program subroutines. In: International Conference on Software Engineering, pp. 795–806. IEEE Computer Society, Washington (2019). https://doi.org/10.1109/ICSE.2019.00087
44. Leroy, X.: Formal verification of a realistic compiler. Commun. ACM **52**(7), 107–115 (2009). https://doi.org/10.1145/1538788.1538814
45. Li, J., Galley, M., Brockett, C., Gao, J., Dolan, B.: A diversity-promoting objective function for neural conversation models. In: Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 110–119. Association for Computational Linguistics, Stroudsburg (2016). https://doi.org/10.18653/v1/n16-1014
46. Lin, C., Och, F.J.: ORANGE: a method for evaluating automatic evaluation metrics for machine translation. In: International Conference on Computational Linguistics, pp. 501–507. Association for Computational Linguistics, Stroudsburg (2004)
47. Liu, K., et al.: Learning to spot and refactor inconsistent method names. In: International Conference on Software Engineering, pp. 1–12. IEEE Computer Society, Washington (2019). https://doi.org/10.1109/ICSE.2019.00019
48. Luong, T., Pham, H., Manning, C.D.: Effective approaches to attention-based neural machine translation. In: Empirical Methods in Natural Language Processing, pp. 1412–1421. Association for Computational Linguistics, Stroudsburg (2015). https://doi.org/10.18653/v1/d15-1166
49. Mahboubi, A., Tassi, E.: Mathematical Components Book (2017). https://math-comp.github.io/mcb/. Accessed 17 Apr 2020
50. Mathematical Components Team: Missing lemmas in Seq (2016). https://github.com/math-comp/math-comp/pull/41. Accessed 18 Apr 2020

51. McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine, part I. Commun. ACM **3**(4), 184–195 (1960). https://doi.org/10.1145/367177.367199
52. Miara, R.J., Musselman, J.A., Navarro, J.A., Shneiderman, B.: Program indentation and comprehensibility. Commun. ACM **26**(11), 861–867 (1983). https://doi.org/10.1145/182.358437
53. de Moura, L., Avigad, J., Kong, S., Roux, C.: Elaboration in dependent type theory. CoRR abs/1505.04324 (2015). https://arxiv.org/abs/1505.04324
54. de Moura, L., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE 2015. LNCS (LNAI), vol. 9195, pp. 378–388. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-21401-6_26
55. Müller, D., Rabe, F., Sacerdoti Coen, C.: The Coq library as a theory graph. In: Kaliszyk, C., Brady, E., Kohlhase, A., Sacerdoti Coen, C. (eds.) CICM 2019. LNCS (LNAI), vol. 11617, pp. 171–186. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-23250-4_12
56. Nanevski, A., Ley-Wild, R., Sergey, I., Delbianco, G., Trunov, A.: The PCM library (2020). https://github.com/imdea-software/fcsl-pcm. Accessed 24 Jan 2020
57. Nie, P., Palmskog, K., Li, J.J., Gligoric, M.: Deep generation of Coq lemma names using elaborated terms. CoRR abs/2004.07761 (2020). https://arxiv.org/abs/2004.07761
58. OCaml Labs: PPX (2017). http://ocamllabs.io/doc/ppx.html. Accessed 23 Jan 2020
59. Ogura, N., Matsumoto, S., Hata, H., Kusumoto, S.: Bring your own coding style. In: International Conference on Software Analysis, Evolution and Reengineering, pp. 527–531. IEEE Computer Society, Washington (2018). https://doi.org/10.1109/SANER.2018.8330253
60. Papineni, K., Roukos, S., Ward, T., Zhu, W.: BLEU: a method for automatic evaluation of machine translation. In: Annual Meeting of the Association for Computational Linguistics, pp. 311–318. Association for Computational Linguistics, Stroudsburg (2002)
61. Paszke, A., et al.: Automatic differentiation in PyTorch. In: Autodiff Workshop (2017). https://openreview.net/forum?id=BJJsrmfCZ
62. Rahman, M., Palani, D., Rigby, P.C.: Natural software revisited. In: International Conference on Software Engineering, pp. 37–48. IEEE Computer Society, Washington (2019). https://doi.org/10.1109/ICSE.2019.00022
63. Raychev, V., Vechev, M., Krause, A.: Predicting program properties from "big code". In: Symposium on Principles of Programming Languages, pp. 111–124. ACM, New York (2015). https://doi.org/10.1145/2676726.2677009
64. See, A., Liu, P.J., Manning, C.D.: Get to the point: summarization with pointer-generator networks. In: Annual Meeting of the Association for Computational Linguistics, pp. 1073–1083. Association for Computational Linguistics, Stroudsburg (2017). https://doi.org/10.18653/v1/P17-1099
65. Serban, I.V., Sordoni, A., Bengio, Y., Courville, A., Pineau, J.: Building end-to-end dialogue systems using generative hierarchical neural network models. In: AAAI Conference on Artificial Intelligence, pp. 3776–3783. AAAI Press, Palo Alto (2016)
66. Sergey, I., Nanevski, A., Banerjee, A.: Mechanized verification of fine-grained concurrent programs. In: Conference on Programming Language Design and Implementation, pp. 77–87. ACM, New York (2015). https://doi.org/10.1145/2737924.2737964

67. Shneiderman, B., McKay, D.: Experimental investigations of computer program debugging and modification. Hum. Factors Soc. Ann. Meet. **20**(24), 557–563 (1976). https://doi.org/10.1177/154193127602002401
68. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: Advances in Neural Information Processing Systems 27, pp. 3104–3112. MIT Press, Cambridge (2014)
69. Suzuki, J., Nagata, M.: Cutting-off redundant repeating generations for neural abstractive summarization. In: Conference of the European Chapter of the Association for Computational Linguistics, pp. 291–297. Association for Computational Linguistics, Stroudsburg (2017). https://doi.org/10.18653/v1/e17-2047
70. Unanue, I.J., Borzeshi, E.Z., Piccardi, M.: A shared attention mechanism for interpretation of neural automatic post-editing systems. In: Workshop on Neural Machine Translation and Generation, pp. 11–17. Association for Computational Linguistics, Stroudsburg (2018). https://doi.org/10.18653/v1/w18-2702
71. Wiedijk, F.: Statistics on digital libraries of mathematics. Stud. Logic Gramm. Rhetor. **18**(31), 137–151 (2009)
72. Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Comput. **1**(2), 270–280 (1989). https://doi.org/10.1162/neco.1989.1.2.270
73. Woos, D., Wilcox, J.R., Anton, S., Tatlock, Z., Ernst, M.D., Anderson, T.: Planning for change in a formal verification of the Raft consensus protocol. In: Certified Programs and Proofs, pp. 154–165. ACM, New York (2016). https://doi.org/10.1145/2854065.2854081
74. Xu, S., Zhang, S., Wang, W., Cao, X., Guo, C., Xu, J.: Method name suggestion with hierarchical attention networks. In: Workshop on Partial Evaluation and Program Manipulation, pp. 10–21. ACM, New York (2019). https://doi.org/10.1145/3294032.3294079
75. Zhang, S., Zheng, D., Hu, X., Yang, M.: Bidirectional long short-term memory networks for relation classification. In: Pacific Asia Conference on Language, Information and Computation, pp. 207–212. Association for Computational Linguistics, Stroudsburg (2015). https://doi.org/10.18653/v1/p16-2034