Passport: Improving Automated Formal Verification Using Identifiers

ALEX SANCHEZ-STERN and EMILY FIRST, University of Massachusetts Amherst
TIMOTHY ZHOU, University of Illinois Urbana-Champaign
ZHANNA KAUFMAN and YURIY BRUN, University of Massachusetts Amherst
TALIA RINGER, University of Illinois Urbana-Champaign

Formally verifying system properties is one of the most effective ways of improving system quality, but its high manual effort requirements often render it prohibitively expensive. Tools that automate formal verification by learning from proof corpora to synthesize proofs have just begun to show their promise. These tools are effective because of the richness of the data the proof corpora contain. This richness comes from the stylistic conventions followed by communities of proof developers, together with the powerful logical systems beneath proof assistants. However, this richness remains underexploited, with most work thus far focusing on architecture rather than on how to make the most of the proof data. This article systematically explores how to most effectively exploit one aspect of that proof data: identifiers.

We develop the Passport approach, a method for enriching the predictive Coq model used by an existing proof-synthesis tool with three new encoding mechanisms for identifiers: category vocabulary indexing, subword sequence modeling, and path elaboration. We evaluate our approach's enrichment effect on three existing base tools: ASTactic, Tac, and Tok. In head-to-head comparisons, Passport automatically proves 29% more theorems than the best-performing of these base tools. Combining the three tools enhanced by the Passport approach automatically proves 38% more theorems than combining the three base tools. Finally, together, these base tools and their enhanced versions prove 45% more theorems than the combined base tools. Overall, our findings suggest that modeling identifiers can play a significant role in improving proof synthesis, leading to higher-quality software.

CCS Concepts: • **Software and its engineering** → *Software verification*; *Formal software verification*; • **Theory of computation** → *Automated reasoning;*

Additional Key Words and Phrases: Proof assistants, proof engineering, proof synthesis, machine learning

ACM Reference format:
Alex Sanchez-Stern, Emily First, Timothy Zhou, Zhanna Kaufman, Yuriy Brun, and Talia Ringer. 2023. Passport: Improving Automated Formal Verification Using Identifiers. *ACM Trans. Program. Lang. Syst.* 45, 2, Article 12 (June 2023), 30 pages.
https://doi.org/10.1145/3593374

This work is supported by the Defense Advanced Research Projects Agency under grant no. HR0011-22-9-006, and by the National Science Foundation under grant no. CCF-2210243.
Authors' addresses: A. Sanchez-Stern (corresponding author), E. First (corresponding author), Z. Kaufman, and Y. Brun, University of Massachusetts Amherst; emails: {sanchezstern, efirst, zhannakaufma, brun}@cs.umass.edu; T. Zhou and T. Ringer, University of Illinois Urbana-Champaign; emails: {ttz2, tringer}@illinois.edu.

0164-0925/2023/06-ART12 $15.00
https://doi.org/10.1145/3593374

1 INTRODUCTION

Verifying software with proof assistants gives engineers the potential to prove the absence of costly and possibly dangerous bugs, leading toward more reliable software systems. Teams of specialized experts have already realized this potential for large and critical systems, such as operating system microkernels [Klein et al. 2009], distributed systems [Wilcox et al. 2015], and compilers [Leroy 2009], among hundreds of other formally verified software systems [Ringer et al. 2019]. These advances have already had significant impact on industry. For example, Airbus France uses the CompCert [Leroy 2009] C compiler to ensure safety and improve performance [Souyris 2014]; Chrome and Android both use cryptographic code formally verified in Coq to secure communication [Erbsen et al. 2019]. But the full potential of these proof assistants still remains far from realized, as the costs of verified software development and maintenance remain high, even for experts [Ringer et al. 2020].

To prove theorems in these proof assistants, proof engineers typically write high-level sequences of strategies called *proof scripts*, which guide the proof assistant toward low-level, machine-checkable representations called *proof objects* [Ringer et al. 2019]. In recent years, techniques that use machine learning to synthesize these proof scripts have shown promise in alleviating some of the effort of verification [First and Brun 2022; First et al. 2020; Paliwal et al. 2020; Sanchez-Stern et al. 2020; Yang and Deng 2019]. These *proof-synthesis* tools learn from corpora of existing proof scripts and theorems to automate the construction of proof scripts for new theorems. In particular, these tools build predictive models of proof scripts, and then use search to explore the proof-script space. This process uses the proof assistant to guide the search and evaluate ultimate success.

In this article, we explore ways of improving these predictive models by better exploiting the richness of the proof data that they learn from. We focus in particular on modeling *identifiers*: the names that uniquely identify theorems, datatypes, functions, type constructors, and local variables. Previous machine-learning-guided proof-synthesis tools have either ignored the names of individual identifiers completely and only encoded basic categorical information about them, or given common identifiers unique indices and marked all others as unknown, without category information. In this article, we develop the Passport approach, which enhances the models used by existing proof-synthesis tools with three new encoding mechanisms for identifiers: category vocabulary indexing, subword sequence modeling, and path elaboration. We implement our approach for tools that synthesize proofs for the Coq proof assistant [Coq Development Team 2021] and show that all three of these encodings improve performance of the end-to-end tool.

The term "Passport approach" refers to our approach of enhancing the model of an existing proof-synthesis tool with identifier information. Most of our evaluation focuses on the application of Passport to a single existing tool, Tok [First et al. 2020]; where unambiguous, we refer to the resulting tool as Passport. Where necessary for clarity, we make explicit the distinction between the approach and the tool resulting from enhancing existing the model of Tok with our approach.

Identifiers in Passport. The Passport approach encodes identifiers with three different encoding mechanisms (described in detail in Sections 3 and 4):

(1) **Category Vocabulary Indexing**: We encode each identifier with the category it comes from (global definition, local variable, or type constructor); and for the most common identifiers in each category, we encode indices corresponding to their names. That is, each common identifier is given a unique tag, associating it with all other uses of that exact identifier.
(2) **Subword Sequence Modeling**: For all identifiers, we use a subword sequence model to draw bridges between related names. That is, identifiers are broken down into common word-pieces and processed with a sequence model.

(3) **Path Elaboration**: For type constructors and global definitions, we encode their *fully-qualified paths* —the names of directories, files, and modules within which they are contained.

While we focus on Coq in this article, similar techniques should apply for other proof assistants, including Lean [Lean Development Team 2021], Isabelle/HOL [Isabelle Development Team 2021], and Agda [Agda Development Team 2021].

Results. We evaluate the Passport approach using the CoqGym benchmark [Yang and Deng 2019] of 124 open-source Coq projects. We compare to three existing search-based proof-synthesis tools, ASTactic [Yang and Deng 2019], Tac, and Tok [First et al. 2020]. We find that all three of our encoding mechanisms improve tool performance, in terms of being able to prove more theorems fully automatically. For example, adding path elaboration leads to proving 12.6% more theorems. We also measure the impact of adding identifier information to each of the categories of identifiers individually, and find that the Passport approach is useful for each.

Together with the three prior tools, tools enhanced with the Passport approach are able to fully automatically prove 1,820 of the 10,782 theorems in our benchmark test set, whereas without the enhancements, these prior tools combined can prove 1,259 theorems. That is an increase of 45% theorems proven over this prior work.

Contributions. The main contributions of our work are as follows:

(1) The Passport approach (Section 4) consisting of a set of techniques for encoding identifiers in a proof assistant context.
(2) The Passport implementation of that approach as a standalone tool within an existing proof-synthesis framework. Passport is open-source: https://github.com/LASER-UMASS/Passport
(3) An evaluation (Section 5) showing that (1) the Passport approach improves proof synthesis when applied to three prior tools, (2) each mechanism for encoding identifiers helps model proof scripts more precisely and improves performance of proof synthesis, and (3) encoding each identifier category alone is still an improvement over not encoding any.
(4) A forward-looking discussion (Section 6) of the challenges that we faced when building Passport (relative to building symbolic proof automation), along with potential solutions to those challenges. Our evaluation includes an experiment measuring the impact of nondeterministic training variance (Section 5.6).

2 BACKGROUND ON PROOFS AND PROOF SYNTHESIS

To write proofs in Coq, the proof engineer starts by stating a theorem to prove. They then write a proof that this theorem holds. Every theorem in Coq is a type definition, described in a rich type system; writing a proof in Coq amounts to finding a term with the stated theorem type.[1]

But doing this directly can be challenging, so instead, proof engineers write these proofs interactively with Coq's help. At each step, proof engineers pass Coq high-level strategies called *tactics*, and Coq responds with the current proof obligations after executing each tactic. Each tactic guides Coq in a search for a term with the stated type, refining the state until no new obligations hold. At that point, the proof engineer has written a sequence of tactics called a *proof script* (like the one in Figure 3(a))—and Coq, for its part, has constructed a *proof term* or *proof object* with the stated type. The language of proof scripts in Coq is called Ltac, and the language of proof terms in Coq, as well as programs and definitions, is called Gallina.

[1]This refers to the Curry–Howard correspondence, which shows type systems and proof systems to be equivalent.

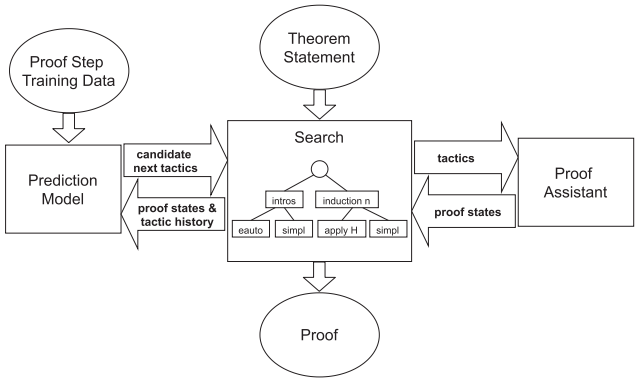


Fig. 1. The system architecture of a machine-learning-prediction-guided proof-synthesis tool.

In recent years, machine-learning-guided proof-synthesis tools have been developed which aim to make the burden of proving easier by automatically generating the proof script, instead of asking the user to write it. While the approaches of these tools can differ, most share similar components and structure.

Figure 1 shows the common architecture of most machine-learning-guided proof-synthesis tools. At the heart of these tools is the prediction model, which guides the proof search by producing *tactic predictions*, or candidate next tactics, at every step. Every prediction model takes as input some set of information about the proof state or proof script, and produces a set of candidate tactics. The tool uses the prediction model to predict one or more likely first tactics and then uses the proof assistant to get feedback on those tactics (e.g., rejecting ones that result in an error or fail to modify the proof state). Then, the tool explores the space of possible proofs by iterating using the prediction model to predict the next tactic and the proof assistant to get feedback and prune the search. As a result, the prediction model's accuracy is critical to the potential success of the search procedure, and for the model to have a chance of being accurate, it must effectively capture the current proof state, and use it to make predictions. The Passport approach works by enhancing the quality of the prediction model, in turn, leading to a better exporation of the proof space.

ASTactic and TacTok. Passport's tactic model architecture inherits the design choices of ASTactic's model [Yang and Deng 2019] for encoding the proof obligations and TacTok's model [First et al. 2020] for encoding the proof script.

Proof obligations consist of the goals to be proven, local context, and the environment. Each term of the proof state has an underlying **abstract syntax tree** (**AST**) representation. ASTactic serializes these ASTs and uses a TreeLSTM [Tai et al. 2015] to encode them [Yang and Deng 2019]. TacTok's model adopts this encoding for the proof state.

The proof script consists of a sequence of tokens in Ltac. Before encoding these tokens, each proof script is preprocessed to remove high-frequency low-signal tokens, such as punctuation. TacTok's model uses a Bidirectional LSTM [Peters et al. 2018] to encode this sequence of tokens [First et al. 2020].

ASTactic's and TacTok's models are trained using supervised learning with a set of human-written proofs to predict the next proof step (tactic and arguments) of an incomplete proof. A limited generative tree-grammar tactic model, adopted from ASTactic [Yang and Deng 2019], makes these downstream predictions. While there may be many valid proofs for a single theorem statement, there is no clear way of determining how appropriate an alternative tactic or proof is, so the model is taught to imitate human-written proofs.

```
Definition posnat¹ := {n² : nat | n > 0}.

Inductive posnatEq¹ : posnat -> posnat -> Prop :=
  | posnatEq_intro³ : ...

Definition posnatMult¹(p1² p2² : posnat) : posnat := ...
```

Fig. 2. Definitions related to the posnat type, a type of pairs of natural numbers and proofs that they are greater than zero. These definitions are found in the Foundational Cryptography Framework,[2] retrieved as part of the Verified Software Toolchain.[3]

3 OVERVIEW OF THE PASSPORT APPROACH

The proof state is made up of many Gallina terms; modeling these terms well is key to producing accurate models. However, previous models have left out much of the essential information about identifiers in terms, when they have encoded identifiers at all. Encoding identifiers well is essential because proof corpora in Coq are rich with identifier information. One reason that identifiers are particularly important in Coq is that Coq has no primitive datatypes; *every* referenced type is an identifier. These names can carry a lot of meaning—and that meaning can be reflected in the names of theorems that refer to them. This article describes and evaluates improvements to identifier encodings in the tactic prediction model.

Categories of Identifiers. To begin to harness the latent information in identifiers, the Passport approach adds three categories of identifiers to the term model. To understand these identifier categories, consider the definitions in Figure 2, from a verified cryptography library.

(1) The identifier `posnat` is a *global definition* (highlighted in `red`[1]), it can be used by data types, functions, theorems, or proof scripts, to reference the globally defined `posnat` datatype.
(2) The identifier `n` is a *local variable* (highlighted in `orange`[2]), as it can be referenced within the local context of this term, but not outside of it.
(3) The identifier `posnatEq_intro` is a *type constructor* (highlighted in `yellow`[3]) as it can be referenced in datatypes, functions, theorems, and proof scripts to construct a new `posnatEq` object.

Appendix A further details these categories of identifiers (global definitions, local variables, and constructor names) and provides intuition through examples for why each category may be useful to encode in a tactic prediction model. Appendix A.4 details the implementation effort required for enriching a model with these three categories of identifiers.

Encodings. Figure 3 shows a proof over these definitions, `posnatMult_comm`. This proof says that multiplication of posnats is commutative, meaning you can switch the order of the arguments and the result will always be the same. Making progress in this proof state requires understanding several things about the identifiers involved.

(1) The `exist` type constructor is a common constructor for sigma (existential) types, and there are specialized tactics (like `exists` and `eexists`) for reasoning with those objects.
(2) The goal type, `posnatEq` is related to `posnats` and equality.
(3) The `Nat.mul` function is defined in the Coq's standard library, whereas `mult_gt_0` is a theorem about it defined in the current project.

[3] https://github.com/adampetcher/fcf.
[3] https://vst.cs.princeton.edu/.

```
Lemma posnatMult_comm¹ : forall p1² p2²,
  (posnatEq (posnatMult p1 p2)
            (posnatMult p2 p1)).
Proof.
  intuition.
  unfold posnatMult.
  destruct p1; destruct p2.
```

(a) A partial proof of `posnatMult_comm`.

```
x : nat
g : x > 0
x0 : nat
g0 : x0 > 0
============================
posnatEq¹ (exist³ (fun n² : nat => n > 0)
                  (Nat.mul¹ x² x0²)
                  (mult_gt_0¹ g² g0²))
          (exist (fun n : nat => n > 0)
                 (Nat.mul x0 x)
                 (mult_gt_0 g0 g))
```

(b) The proof state at this point in the proof.

Fig. 3. A proof using the definitions in Figure 2, from the same file.

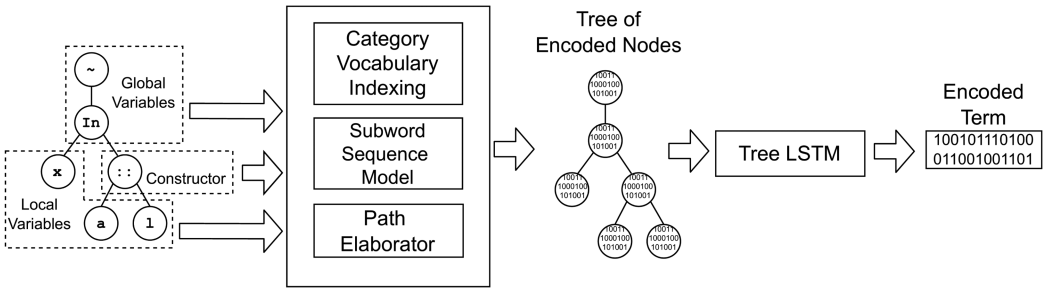


Fig. 4. The architecture of Passport's identifier processing.

Understanding these things requires three different approaches: attaching special signifiers to common identifiers, processing the individual pieces of identifiers to understand where they connect to different concepts, and remembering where the definitions being referenced are defined.

The crux of this article is the enrichment of a proof-synthesis model for Coq with rich information about identifiers. Figure 4 shows an overview of how the Passport approach encodes identifiers. To fully take advantage of the richness of these identifiers, our design employs three key encoding mechanisms:

(1) *Category Vocabulary Indexing* (Section 4.1), which separately considers different kinds of common identifiers in a proof development,
(2) *Subword Sequence modeling* (Section 4.2), which draws bridges between all identifiers, and
(3) *Path Elaboration* (Section 4.3), which encodes the location where the object referred to by each identifier is defined.

Category vocabulary indexing allows us to assign unique labels to common identifiers in the code. In this case, that means giving a unique label to the `exist` type constructor, so that we can use knowledge from previous proofs which used that precise constructor. Subword sequence modeling allows us to break identifiers up into common pieces, and process those pieces with a sequence model. In this case, that means breaking the `posnatEq` identifier into the chunks `posnat` and `Eq`, so that we can use knowledge from previous proofs that had identifiers with similar pieces. Finally, path elaboration allows us to consider the directories, files, and modules in which the object referenced by the identifier is defined. Here, that means understanding that the multiply identifier refers to a function defined within `Coq.Init.Nat`, but the `mult_gt_0` refers to a lemma defined in the current file.

Armed with the knowledge from these three encoding mechanisms, our model has everything it needs to suggest tactics that the tool can use to complete the proof of `posnatMult_comm`.

4 PASSPORT ENCODINGS

Identifiers are proxies for semantic information not by accident, but *by design*. By taking advantage of the information in identifiers, term models can learn from the design principles the proof engineer has already followed to make proof developments easier to read, understand, and build on. To extract this information from identifiers, the Passport approach uses three encoding mechanisms: **category vocabulary indexing** (Section 4.1), **subword sequence modeling** (Section 4.2), and **path elaboration** (Section 4.3).

In the implementation, Passport uses Coq query commmands to access the full Coq environment when extracting identifier information, so it is not limited to any particular subset of the environment.

4.1 Category Vocabulary Indexing

In each identifier category (global definitions, local variables, and type constructors), there are many common identifiers used across proof developments. These identifiers are so common that we can learn a significant amount about how to understand them from their previous uses. For instance, in the example from Figure 3, the `exist` type constructor is part of the standard library, and many proofs in our training data reason with it. Even when an identifier is not very common, we can still understand a lot about it by knowing what category it is in.

To take advantage of these properties of identifiers, we developed **category vocabulary indexing**. This encoding mechanism tags every identifier with the category it comes from and, if the identifier is commonly used enough, a unique tag for that particular identifier. By giving common identifiers a unique tag, we can generalize across their many appearances, and predict tactics that worked well with them in the past. And by marking identifiers with their category, either global definition, local variable, or type constructor, we can disambiguate identifiers with the same name from different categories, and learn useful information about even uncommon identifiers.

The models in some previous tools for machine-learning-guided proof-synthesis, such as Proverbot9001 [Sanchez-Stern et al. 2020] and Tactician [Blaauwbroek et al. 2020], use vocabulary indexing for common identifiers, but make no category distinctions. This is a reasonable approach, because in Coq, the names of global definitions, local variables, and type constructors share a common namespace. However, in the Passport approach, we decided to distinguish between identifiers of different categories, in part because manual analysis of the training data revealed different naming conventions for different categories. For example, single-letter identifiers seemed to almost exclusively represent local variables, with uppercase for types (like `A` in Figure 10), and lowercase for terms (like `x` in Figure 3); longer uppercase identifiers generally refer either to sort names (like `Set` or `Prop`) or type constructors (like `Some` or `None`). This means that when human provers see an identifier, even if they have not seen it before, they often have a sense of what category it belongs to.

The models in other previous tools for machine-learning-guided proof-synthesis, such as ASTactic and TacTok, make category distinctions, but do not index vocabulary. We learned early on that the possibility of performance regression due to uninformative local variables like `x` had concerned the ASTactic authors, and contributed to their decision not to encode identifiers.[4] However, upon closer inspection of the data, we determined that even when a particular name does not always refer to the same definition, common names can carry information of their own. For instance, variables named `hd` and `tl` consistently refer to the head and tail of a list. These names, too, can benefit

[4] https://github.com/princeton-vl/CoqGym/discussions/60.

from a unique tag which generalizes across their usages. Our manual inspection determined that this can often hold even for single-character variable names.

Implementation. To decide which identifiers are common enough to be indexed, we use our training dataset to create a fixed identifier vocabulary. That is, we count the occurrences of each identifier, and include in our vocabulary those whose count is above an experimentally chosen, fixed threshold (see Section 5.7 for an evaluation of different thresholds). Using separate vocabularies for each category of identifier allows us to use different thresholds across different categories; since type constructors are less common overall than local variables, they might require having a lower threshold for being included in the vocabulary.

4.2 Subword Sequence Modeling

Identifier information can be useful not just for learning about individual datatypes, theorems, and functions, but also for drawing bridges between them. Developers often organize development using parts of names to group theorems and functions which refer to common definitions. It turns out these naming conventions can be useful to a model, too.

Many variable names are not simply single unique words, but are made up of multiple parts. These parts could be multiple english words in camel case, such as the case in something like `firstItemInList` broken into "first", "item", "in", and "list". Or they could be components of a single word that carry individual meaning, like `prelocalizations` broken into "pre" , "local" , "ization" , and "s". By breaking these identifiers into pieces, a model built using the Passport approach can learn the meaning of shared pieces and generalize across identifiers.

In the example from Section 3, Passport breaks `posnatMult` into `[pos, nat, Mult]`; with a different subword vocabulary, from a different set of variable occurrences in the training data, it might produce `[posnat, Mult]`. These tokens are processed with a sequence model, so that the identifier's ultimate feature vector reflects the fact that the identifier relates to the "posnat" type, and that it primarily relates to the multiplication operation.

To get a sense for this, let us consider another example. The Coq standard library includes operations about the real numbers `R`, like addition:

```
Rplus¹ : R → R → R.
```

The library contains proofs of theorems about `Rplus`, like this proof (highlighting just one `Rplus` for presentation):

```
Lemma Rplus_eq_compat_l : ∀ (r r1 r2 : R),
    r1 = r2 → Rplus¹ r r1 = Rplus r r2.
Proof.
  intros r r1 r2.
  apply f_equal.
Qed.
```

which proves the theorem that right addition preserves equality.

Suppose we wish to prove the analogous theorem about the natural numbers `nat`, using the addition function `plus` defined over `nat`. We can do this the same way:

```
Lemma plus_eq_compat_l : ∀ (n n1 n2 : nat),
    n1 = n2 → plus¹ n n1 = plus n n2.
Proof.
  intros n n1 n2.
  apply f_equal.
Qed.
```

simply renaming the local variables for style (though the original proof with `r`, `r1`, and `r2` also works with no changes).

The fact that `Rplus` and `plus` are related is explicit in the identifier names: `Rplus` behaves like `plus` over `R`. A model that can draw connections between `plus` and `Rplus` can in some cases reuse proofs about one to derive analogous proofs about the other.

The key here is subword sequence modeling which excels at drawing connections between related words [Gage 1994; Sennrich et al. 2016]. Subword sequence modeling allows us to break the identifier `Rplus` into the chunks `R` and `plus`, and index them separately, connecting them to the identifier `plus`. By drawing these connections, we expect that a model can suggest `intros` and `f_equal` in the body of `plus_eq_compat_l`, by connecting the hypothesis `plus n n1 = plus n n2` to the hypothesis `Rplus n n1 = Rplus n n2`. With subword sequence modeling, the model can learn all of this with no need for semantic information about what each of the reals and naturals represent, or how their addition functions are related.

In the Passport approach, identifiers are broken into subwords using a **byte-pair encoding** (**BPE**) algorithm [Gage 1994; Sennrich et al. 2016], an algorithm that has seen success in code completion models for program synthesis [Karampatsis et al. 2020; Svyatkovskiy et al. 2021]. The algorithm uses the training corpus to make a list of common subwords by starting with a vocabulary of single characters, and iteratively merging common pairs. Then, each identifier is tokenized by greedily consuming the longest matching vocabulary element.

The Passport approach incorporates these tokens as embeddings in a syntax model. Program syntax can generally be modeled in two ways. The simplest way is to model it as an unstructured sequence of words (or more generally, tokens). The alternative is to parse the syntax into a tree, and use a tree based model to process it. One of the advantages of the former is that you can tokenize strings in a number of different ways, including with multiple tokens per identifier (subword tokenization). However, our implementation of Passport builds on a parsed-tree-based model, so there is no existing string tokenizer that could be used for subword tokenization. Instead, we embed a sequence model *within the leaves* of the tree-based syntax model. This means that our subword sequence model only learns how to combine parts of an identifier into a fixed embedding for the identifier, and does not need to learn about other parts of program syntax.

With our category vocabulary indexing, we used separate vocabularies for identifiers of different categories. However, proof developments sometimes demonstrate connections between identifiers from different categories. These connections are lost in using separate vocabularies, so subword encoding is used to maintain these connections. The Passport approach uses a single subword vocabulary, derived from the global variable corpus, to encode identifiers from all categories.

Implementation. There are several subtleties to the implementation of our subword tokenization algorithm, and the BPE which generates its vocabulary. Sometimes there were several possible ways to implement the approach; in general, we made our choices based on the performance of the resulting tool on our benchmarks.

As indicated by the name, byte-pair tokenization often starts with a vocabulary of bytes, not characters, to allow a reasonable base vocabulary size when working with unicode. However this has the downside of sometimes indicating that two identifiers are similar because they share bytes within a unicode character, even if no characters are in common. In our implementation, we use characters as our base vocabulary. To keep our base vocabulary of a reasonable size, we only include those characters which are present in the training corpus. Since Coq programmers generally only use a small subset of possible unicode characters, this works well. However, there are in rare cases unicode characters present in the test data which are not present in the training data. To address this, our subword tokenizer drops characters which are not present at all in the vocabulary; this behavior can be changed with a flag to instead produce a special `<unknown>` element.

Many different neural architectures have been used to process sequences of tokens. For language modeling, the most effective models are often those with attention and forgetfulness mechanisms, to capture the long-range dependencies present in text. However, the identifiers we work with are generally short, often only a few subwords long, so we instead use the simplest sequence model, a Recurrent Neural Network, without any attention mechanism.

As with any sequence-based model, there is a question of how to cap the size of sequences so that their lengths can be normalized. With Passport, we found empirically that capping at four tokens per identifier during training, but eight tokens per identifier when synthesizing proofs, is most effective on our evaluation suite. Four subwords is enough to encode the entire name of 98.74% of identifiers in our training data, and eight subwords is enough to encode the entire name 99.97% of the time.

We trained the subword encoder end-to-end alongside the rest of the term encoder and tactic decoder, so that the encoder is trained to retain information about subwords particularly relevant to the task of tactic prediction.

4.3 Path Elaboration

The final encoding mechanism in the Passport approach is path elaboration: the encoding of fully-qualified paths of different identifiers. By paying attention to the fully-qualified paths of different identifiers, the tools using the Passport approach can take advantage of any grouping of identifiers into common modules and files already used by Coq developers to organize development. Tools using Passport approach can also capitalize on proof development styles that dispatch proofs for entire classes of related theorems using powerful tactics—a proof development style recommended by, for example, the popular Coq textbook Certified Programming with Dependent Types [Chlipala 2013].

To gain some intuition for what this means in action, consider this proof of a theorem from the Coq standard library:

```
Theorem not_in_cons A (x a : A) (l : list A):
  ~ In x (a::l) ↔ x<>a ∧ ~ In x l.
Proof.
  simpl. intuition.
Qed.
```

The proof of `not_in_cons` goes through by just two tactics: `simpl` and `intuition`. The `simpl` tactic simplifies the initial goal (no assumptions, with the theorem type as the sole proof obligation) to make it easier to reason about, producing this proof state:

```
A : Type
x, a : A
l : list A
______________________________________(1/1)
~ (a = x ∨ In x l) ↔ x <> a ∧ ~ In x l
```

In this case, the `simpl` tactic has unfolded the `In x (a::l)` on the left side of the identifier into `(a = x ∨ In x l)`.

But the resulting goal is still a bit complex because it chains together a number of logical connectives: if and only if (↔), negation (~), inequality (<>), conjunction (∧), and disjunction (∨). So the `intuition` tactic breaks down logical connectives into simpler subgoals, and dispatches each subgoal automatically.

Taking a step back, it is natural to wonder how the proof engineer could have known to use the `intuition` tactic to dispatch the remaining goals. Intuitively, it made sense to use `intuition` here

```
not_in_cons¹
: ∀ (A² : Type) (x a² : A) (l² : list A),
    Coq.Init.Logic.iff¹
      (Coq.Init.Logic.not¹
        (In¹ A x (cons³ A a l)))
      (Coq.Init.Logic.and¹
        (Coq.Init.Logic.not
          (Coq.Init.Logic.eq¹ A x a))
        (Coq.Init.Logic.not (In A x l))).
```

Fig. 5. The theorem statement not_in_cons, elaborated with paths. Highlighted using the same conventions as in Figure 2, with other paths omitted for brevity.

because the goal consisted of simple statements linked by logical connectives, which intuition excels at. It turns out that the fact that these operators are logical connectives is explicit in the paths of the identifiers in the goal—they all reside in the Coq.Init.Logic module—so we can pass it on to our models by encoding paths.

We can see this by expanding the paths of the identifiers in the theorem statement of not_in_cons (Figure 5). All of the operators in not_in_cons are syntactic sugar for identifiers, which themselves refer to types defined inductively in Coq. For example, conjunction (∧) refers to the inductive type and in the path Coq.Init.Logic. Internally, Coq stores the elaborated theorem with all of these identifiers (like and) and their fully-qualified paths (like Coq.Init.Logic) explicit. Inspecting the elaborated version of not_in_cons shows that the fact that these are logical connectives requires no semantic understanding to deduce—it is explicit in the grouping of identifiers in the Logic module.

We determined that a simple way to pass this intuition on to our models was to encode each of the file and module names inside of fully-qualified paths, taking advantage of the organization of large proof developments to infer tactics used to dispatch related goals.

Implementation. To implement this, we created a dedicated vocabulary and corresponding <unknown> token for file and module names inside of fully-qualified paths, much like we did for each category of identifier. We then used this vocabulary for encoding paths.

As with identifiers, Coq includes fully-qualified paths inside of the ASTs by default, but TacTok and ASTactic had erased those paths from the AST. For example, in Figure 12, the fully-qualified path Coq.Init.Datatypes of the option inductive type shows up in the AST as a directory_path node, with data [Datatypes; Init; Coq].

Elaborating paths was thus similar to adding each of the categories of identifiers: First, we modified the post-processing code to avoid erasing paths. Then, we built a separate vocabulary for common files or modules that paths consisted of, like Datatypes, Init, and Coq in Figure 12. We then encoded each file or module along the path separately, mapping to a dedicated <unknown> token for files or modules in paths that occurred less frequently than the chosen threshold.

5 PASSPORT EVALUATION

We evaluated Passport's ability to successfully prove theorems using the CoqGym benchmark [Yang and Deng 2019], following the evaluation methodology used by several recent papers [First and Brun 2022; First et al. 2020; Yang and Deng 2019].

In summary, our results show:

— **The Passport approach improves proving power.** By comparing to previous tools—ASTactic and the two base tools, Tac and Tok, that make up TacTok—we measured additional proving power provided by the Passport approach's encoding of identifiers. The combined

proving power of the tools enhanced by the Passport approach exceeds that of the original tools by 38%, and combining both the enhanced and un-enhanced tools outperforms the combined un-enhanced tools by 45% (Section 5.2).
— **Identifiers improve performance.** All three categories of identifiers improve performance, in aggregate proving 64% more theorems than the individual un-enhanced tool (Section 5.3).
— **All three encoding mechanisms improve performance.** All three categories of identifiers in the Passport approach improve performance in Passport with each of the three encoding mechanisms (Sections 5.4 and 5.5).
— **Our results are meaningful beyond variance introduced by nondeterminism.** Proof synthesis success rate varies by 0.4% for individual tools, and combining many varying runs can improve results by 22% (Section 5.6).
— **Hyperparameter choices impact performance.** We choose our hyperparameters experimentally based on these results (Section 5.7).

All our experiments are affected by nondeterminism, and while the bulk of our experiments only use a single trial, Section 5.6 explores the effect on nondeterminism on the variance of our results and argues that that effect is small.

5.1 Experimental Setup

Benchmark. The CoqGym benchmark includes 124 open-source Coq projects, split into three sets. For our evaluation, we trained on 97 projects (containing a total of 57,719 theorems) and synthesized proofs for 26 projects (containing a total of 10,782 theorems). We exclude one project, coq-library-undecidability, from our evaluation because TacTok's evaluation [First et al. 2020] was unable to reproduce prior results for ASTactic's performance [Yang and Deng 2019] on that project due to internal Coq errors when processing the proof scripts.

Projects in the CoqGym benchmark are a mixture of mathematical formalizations, proven correct programs, and Coq automation libraries. They include several compilers of varying sizes (such as CompCert [Leroy 2009]), distributed systems (such as Verdi [Wilcox et al. 2015]), formalizations of set theory, and more. Some of the projects in CoqGym (such as the automation libraries) do not contain any proofs, but we included them for completeness.

Machines. We ran this paper's experiments using two clusters: a GPU cluster for training and a CPU cluster for synthesizing proofs.

Each node in the GPU cluster has between two and eight NVIDIA GPU cards. There are 4 nodes with 2 NVIDIA Tesla V100 GPUs, and 33 nodes with 8 NVIDIA RTX 2080ti GPUs. The nodes in the GPU cluster all run on a shared ZFS file system, run CentOS Linux, and use Slurm for job scheduling and resource management. We found that training time varied between 12 and 14 hours per epoch, and did not differ significantly between the Passport implementation and the baseline model.

Each node in the CPU cluster has between 24 and 36 cores, with 4 hyperthreads per core. There are

— 1 head node with 24 cores of Xeon E5-2680 v4 @ 2.40GHz, 128GB RAM, and 200GB local SSD disk.
— 50 compute nodes with 28 cores of Xeon E5-2680 v4 @ 2.40GHz, 128GB RAM, and 200GB local SSD disk.
— 50 compute nodes with 28 cores of Xeon Gold 6240 CPU @ 2.60GHz, 192GB RAM, and 240GB local SSD disk.
— 5 compute nodes with 56 cores of Xeon E5-2680 v4 @ 2.40GHz, 264GB RAM, and 30TB local disk.

The nodes in the CPU cluster also all run on a shared ZFS file system, run CentOS Linux, and use Slurm for job scheduling and resource management. The average inference time for a random sample of generated proofs was 0.4 seconds per tactic for the Passport implementation, compared to 0.3 seconds for the baseline model.

Experimental Parameters. Passport attempts to synthesize each proof for a preset amount of time, timing out if it fails to reach `Qed` in that time. Our evaluation used 10 minutes for this timeout, following the choice made by ASTactic [Yang and Deng 2019] and TacTok [First et al. 2020]. Following a design decision made by ASTactic, we limited our search to a total of 300 attempted tactics, and restrict solutions to be no longer than 50 tactics long. Our experiments use 200 as the default category vocabulary threshold (recall Section 4.1) and 4,096 as the default byte-pair merge threshold (recall Section 4.2). We use 128 as the default vector dimension for term, grammar, and terminal/non-terminal symbol embeddings, as well as the dimension of the LSTM controller. For all other parameters, we follow those used by ASTactic [Yang and Deng 2019] and TacTok [First et al. 2020].

Implementation. Overall, the Passport approach implementation is 1.5K lines of code and took four developers about a year to build. While the conceptual and design aspects of the Passport approach can extend to all prediction-model-driven, search-based, proof-synthesis tools, the current implementation is straightforwardly applicable to all such tools built within the CoqGym environment [Yang and Deng 2019].

This implementation adds three embeddings for category indexes and one for paths, with 428, 136, 27, and 262 items for global definitions, locals, constructors, and paths, respectively. This results in a corresponding increase to the first layer of Tok's term encoder. The new subword embedding contains 4,164 items and is encoded with an RNN using a hidden size of 32. When implementing these new model components, we optimized for simplicity over model size, so we believe that the model size could be decreased further without significantly impacting accuracy.

The original ASTactic, Tok, and Tac models used a 256-float symbol embedding size. However, we observed no significant difference between those models using a 256-float symbol embedding, and using a 128-float symbol embedding. As a result, our model uses 128-float symbol embeddings, and, where appropriate, we compared to versions of other models with a 128-float symbol embedding. Overall, these changes to model size had no significant impact on training time, as described above.

5.2 The Passport Approach's Effect on Proof-Synthesis Tools

In this section, we show that the addition of our identifier information improves the end-to-end performance of proof search tools. Since Passport is implemented in the ASTactic/TacTok framework, we were able to evaluate our changes against three base tools: an ASTactic-like[5] tool, Tac, and Tok. ASTactic was developed as part of the CoqGym project [Yang and Deng 2019] and uses only proof contexts as input to their prediction model. By contrast, the models in Tac and Tok (developed as part of the TacTok project [First et al. 2020]) additionally model the proof script up to the current point, with Tac's model encoding the tactics in the proof script, and the Tok's model encoding all the tokens except punctuation in the proof script.

Figure 6 shows the results of adding identifier information to all three of these tools. Adding identifiers to each of the three tools significantly improves their ability to prove theorems. Adding

[5]We were not able to replicate the original results of ASTactic [Yang and Deng 2019], so for our evaluations we trained a model with the same embedding vector dimensions as our own models. For this reason, we are using the term ASTactic-like when we describe our results.

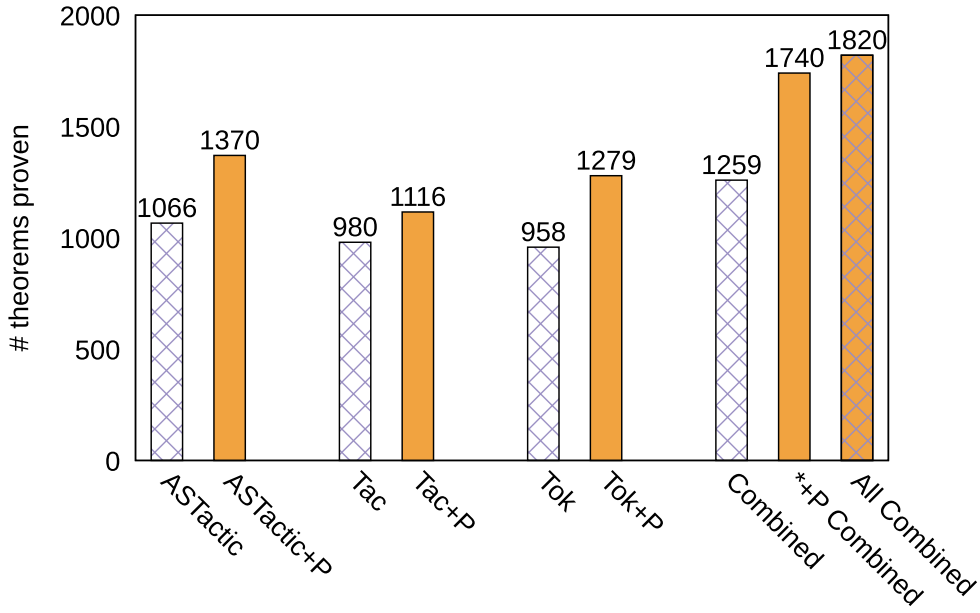
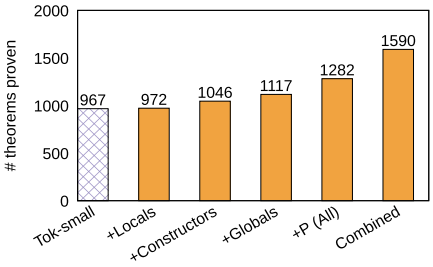


Fig. 6. The effect of adding all of the three encodings for three identifier types to several proof-synthesis tools. The purple crosshatch bars represent baseline tools based on ASTactic, Tok, and Tac. The orange bars represent our new contributions. The rightmost crosshatch bar, labeled "Combined", is the number of theorems successfully proven by *at least one* of the baseline tools. The orange bar next to that, labeled "*+P Combined", is the number of theorems successfully proven by *at least one* of the tools enhanced by the Passport approach. Finally, the orange *and* crosshatched bar on the far right is the number of theorems proven by at least one of all the presented tools.

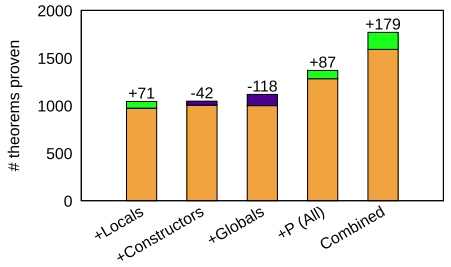
identifier information improves our ASTactic-like tool by 29% (304 additional theorems proved), Tac by 14% (136 additional theorems proved), and Tok by 33% (318 additional theorems proved).

Following TacTok's [First et al. 2020] and Diva's [First and Brun 2022] evaluations, we also explore how the differences in theorems proven by multiple tools lead to more theorems proven overall, and how adding identifier information increases that improvement. When we compute the union of the theorems proven by all our tools enhanced by the Passport approach, and compare that set to the union of the theorems proven by the base tools, we find an improvement of 38%. Comparing the union of theorems proven by all the tools to the union of theorems proven by the three base tools, we find an improvement of 45%.

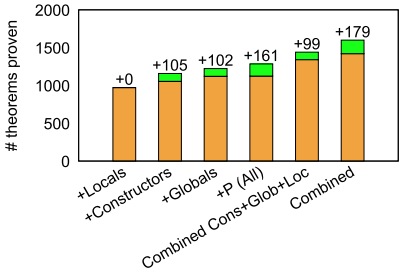
Next, we examine the complexity of the proofs that Passport generated. Using human-written proof-script length as a rough proxy for complexity, we note that Passport successfully synthesized proof scripts for 351 theorems for which the human-written proof scripts were at least 5 tactics long. For 54 of those theorems, the human-written proof scripts were at least 10 tactics long. This observation suggests that Passport is able to synthesize a significant number of nontrivial proofs. For 280 theorems, Passport was able to synthesize proof scripts that were shorter than the human-written ones. In one particular case, the human-written script was 139 tactics long, while Passport's script was only 2 tactics long. The baseline tool produced 239 proofs for which the human-written proof scripts were at least 5 tactics long, so Passport proved 46.9% more theorems with human-written proofs of that length. For theorems with human-written proofs of length 10 or more, the baseline tool produced 37 proofs, so Passport proved 45.9% more such



(a) The impact of category vocabulary indexing on three identifier categories (without subwords or paths): local variables, type constructors, and global definitions.



(b) The impact of subword encoding on each of the categories of identifiers (with category vocabulary indexing but without paths).



(c) The impact of fully-qualified path encoding of type constructors and global definitions (with category vocabulary indexing but without subwords).

Fig. 7. The impact of various encoding techniques on theorems proven.

theorems. Finally, the baseline model produced proofs shorter than the human-written proofs for 171 theorems, so Passport did so for 63.7% more theorems.

Examining the time it takes Passport to synthesize a proof script, the successfully generated proof scripts took between 0.08 and 86.6 seconds to generate, with the mean of 2.9 seconds.

5.3 Identifier Categories

In the Passport approach, we model three categories of identifiers. While the experiment in Section 5.2 showed that modeling identifiers from these categories are effective together, we also want to show the utility of the identifier categories individually.

Figure 7(a) shows the individual results of just adding local variables, type constructors, and global definitions. For consistency, this experiment compares to a Tok-like tool with a model with smaller embedding sizes, as Passport uses that model to add identifier information to.

Each of the identifier types added individually increases the number of theorems proven, though the increase from local variables alone is marginal. Adding type constructors alone proves 8% more theorems than the baseline, adding global definitions alone proves 16% more theorems, and adding local variables alone proves 0.5% more theorems.

However, no identifier category added individually is close to the impact of adding all three. Adding all three identifier types, without subword information, proves 33% more theorems.

Finally, though none of the tools with individual identifier types prove as many theorems as the one with all of them together, some of these individual identifier-enriched tools prove theorems that the all-identifiers-enriched tool does not. The union of the theorems proven by the individual identifier-enriched tools and the all-identifiers-enriched tool contains 64% more theorems than the baseline tool.

These experiments show that each identifier category is useful for producing a more effective proof-synthesis tool, and that the identifier categories help with a diverse set of theorems, so combining the results of adding different subsets of identifiers helps further.

5.4 Subwords

Figure 7(b) shows the impact of adding subword encodings to our identifier-enriched tools (Section 4.2). Adding the subword encoding does not benefit all types of identifiers individually. In fact, it makes two (type constructors and global definitions) out of the three identifier categories perform worse than when those identifiers are used individually, possibly due to overfitting.

However, when subwords are added to the full tool with all the identifier categories, they improve results by 7%. This improvement is greater than what the cumulative impact of adding subwords to individual identifier-enriched tools, suggesting that subwords particularly help with making connections between multiple identifier types. In fact, even though subword sequence modeling does not help global definitions alone, when global definitions are combined with the other identifier types, removing subword encoding significantly hurts results.

The most likely explanation for these results is that for subwords to be effective, a sufficiently large number of identifiers is necessary to encounter a non-trivial number of repeated subwords, allowed for learning semantics of those subwords. Adding subwords to only a single type of identifier likely does not meet that threshold, but using all identifiers leads to a significant improvement in the tool's proving power.

5.5 Paths

Figure 7(c) shows the impact of removing path elaboration (Section 4.3) from various identifier types in the Passport model. Since local variables do not have paths, there is no impact of removing path elaboration. Subwords were not included in this experiment, as we wanted to isolate the impact of paths.

Path elaboration benefits both type constructors and global definitions: increasing theorems proven for type constructors alone by 10% and increasing theorems proven for global definitions alone by 9%. The union of the theorems proven using these categories alone and the theorems proven with local variables alone (for which the paths improvement is 0%) is 7% larger than without path elaboration. However, when we add path elaboration to Passport's model with *all three* identifier categories, it increases the number of theorems proven by 12.6%.

These results indicate that the impact of adding path elaboration to a model that implements local variables, type constructors, and global definitions is greater than the combined effect on individual models. Similarly to the subword experiment above, these results suggest that encoding fully-qualified paths helps connect identifiers across categories; learning about how type constructors from a particular module behave helps in dealing with global definitions from that module, and visa versa. However, unlike the subword experiment, paths seem to benefit all identifiers for which they are implemented individually as well as in combination.

5.6 Nondeterministic Model Variance

During the course of our evaluation, we found that models trained in the ASTactic framework had significant variance in their downstream proof-synthesis success rate, even when the model code

and training data were identical. While part of this variance could be attributed to different hardware and other hard-to-control factors (see Section 6), even when controlling for all those factors, there was still variance. After months of investigation, we found that the cause was nondeterminism at the hardware and framework level, some of it undocumented [Gao 2022; Reichel 2022].

Nondeterminism in model training is not specific to proof search, and has in fact been documented in the ML community at large [Pham et al. 2020a; Qian et al. 2021; Shamir and Lin 2022]. However, it is not immediately obvious how these effects would impact proof search, since they are usually measured as inaccuracy in the top prediction of a model, while proof-search tools generally use multiple model predictions, smoothing out some inaccuracy.

To measure the impact of nondeterministic training variance on proof search, we trained our model with identifiers added to Tok's model 20 times. On average, the tool using one of these models proved 11.9% (1,279 theorems), with the maximum proving 12.0% (1,294 theorems) and the minimum proving 11.6% (1,256 theorems). The 0.4% spread (38 theorems) shows that training the same model can lead to small differences in overall success rates. Our result for adding local variables alone (with no other identifiers) and without subword encoding is within this variance range. However, the impact of local variables is better captured with the addition of subwords and together with other identifiers, which yields results significantly outside of this range.

Interestingly, the union of the theorems proven by the tool using these 20 models is 14.5% (1,564 theorems), an improvement of 22% over the average. This demonstrates that the scale of the differences in *which* theorems model-based tools can prove as a result of nondeterministic training variance is much larger than the scale of the differences in *how many* they prove. Thus, the variance from training nondeterminism serves as a dimension for model diversity, which can be used to improve proof synthesis, similarly to the approach taken by Diva [First and Brun 2022].

5.7 Hyperparameters

As discussed in Section 4.1, each of the identifier types we add has a vocabulary of the most common identifiers of that type, giving a fixed encoding of those identifiers in addition to the subword encoding. We count the occurrences of the identifiers in the training set to determine which identifiers occur more than a specified threshold, and then only include those identifiers in our vocabulary. For example, if we have a threshold of 100, then all the identifiers that occur at least 100 times in the training set will be included in the vocabulary. That threshold is a hyperparameter that we can vary for each type of identifier, and it determines the size of the vocabulary.

Figure 8 shows the performance impact of different values of that hyperparameter for different identifiers. As you can see, the performance of various vocabulary sizes for global definitions, local variables, and type constructors is all fairly jagged, though they all peak at around 200 occurrences, which we set as the default in the rest of our experiments.

It is interesting to note that, while the thresholds which produce the best results are the same for the different identifier categories, this results in drastically different vocabulary sizes: 427 global definitions meet the threshold, but only 135 local variables and 26 type constructors do. This justifies our decision to use a fixed occurrence threshold to pick vocabulary rather than using the n most common identifiers from each category.

However, there are signs that our method of picking vocabulary to index could be improved. Sometimes, adding identifiers with fewer occurrences, such as the global definitions with between 180 and 200 occurrences, helps; while adding those with more occurrences, such as the global definitions with between 200 and 220 occurrences, hurts. This suggests that the number of occurrences does not monotonically predict the usefulness of indexing a particular identifier, even though it is the most common approach. Future systems should investigate new metrics to pick vocabulary for indexing. Finally, these experiments indicate that the model—and therefore the

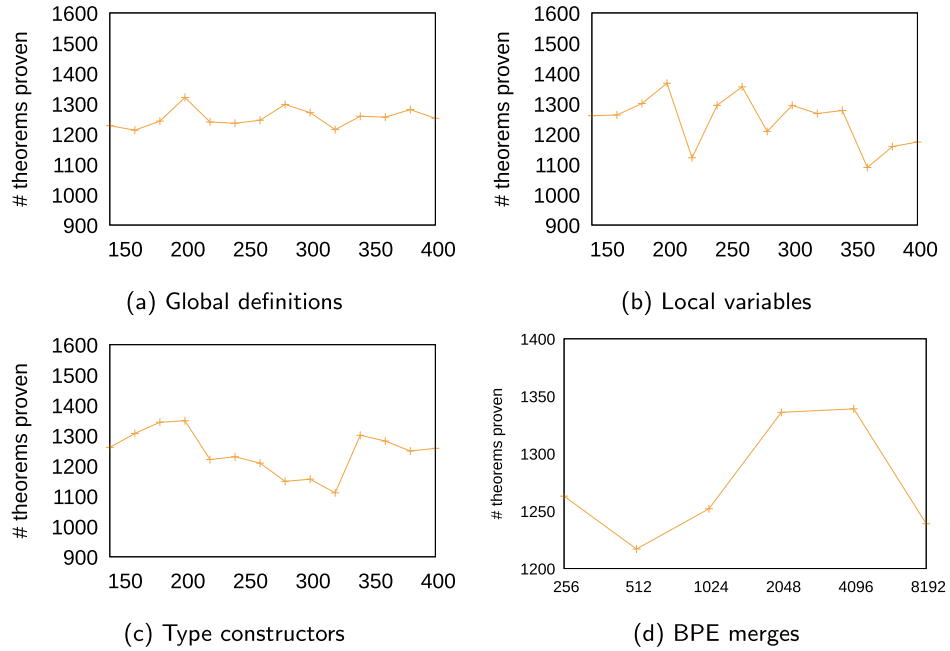


Fig. 8. The impact of different vocabulary thresholds for the various categories of identifiers. A smaller threshold means the vocabulary is larger.

proof-search tool—is sensitive to small changes in hyperparameters, similar to how model-based tool performance varies greatly from nondeterminism at the hardware level in model training.

The subword encoding we use also has several hyperparameters which can be varied; principle among these is the number of byte-pair merges, which determines the size of the subword vocabulary. Figure 8(d) shows the effect of different subword vocabulary sizes on success rate. The default byte-pair merge threshold of 4,096 is represented as the highest point on the graph.

6 DISCUSSION

We believe that it is prudent to broaden the discourse around machine learning for proofs to consider not just the tool produced, but also the development processes in building these tools. It is for this reason that we step back and discuss our experiences, centering challenges that we encountered in three areas: the feedback cycle, reproducibility, and debugging.

Feedback Cycle. The feedback cycle for developing Passport was slow. Every time we changed an encoding, we had to retrain the model, a process that took around two days. Mistakes in the code or in the training parameters would often not manifest until evaluation, at which point we would need to retrain once more. This slow feedback cycle quickly added up, so that even a small change could take weeks.

In traditional supervised learning, training dominates development time, as evaluating a model means running it just once on the test set. However, in the context of proof search, evaluation on a large benchmark set often takes as many or more computational resources as training, though it is usually more parallelizable across machines.

In the machine learning literature, techniques have been proposed to make training faster [Lepikhin et al. 2020; Li et al. 2022b; Popel and Bojar 2018; Rajbhandari et al. 2020], which could be

directly applied in proof search. And more tooling like data trackers [Biewald 2020], data validation, and static types can help catch bugs sooner, resulting in fewer training runs needed during development. Finally, some work in combining multiple models [First and Brun 2022] has shown an ability to speed up proof search, and other search optimizations could also shorten that part of the feedback cycle.

Reproducibility. As discussed and measured in our evaluation (Section 5.6), many current learning frameworks and APIs behave nondeterministically, resulting in nondeterministic variance in our end-to-end proof results. Much of the nondeterminism we encountered is difficult but possible to control, when it stems from hardware differences, random seeds, or OS-level file ordering. However, even when controlling for those factors and all documented nondeterminism, we found our model training was still nondeterministic. During the course of our development, we discovered some PyTorch APIs that were documented as deterministic behaved nondeterministically; we reported that bug, and the developers marked it as high-priority.[6]

A recent paper found this variance in performance across identical training runs to be pervasive in an evaluation of six popular neural networks on three datasets [Pham et al. 2020b]. This article found that very few of the researchers or practitioners surveyed in were aware of possible nondeterminism in these systems. We recommend that future researchers using machine learning for proof search document the hardware and software used to train, and report some measure of the variance in their models results.

Debugging. The debugging of systems that mix machine learning and symbolic manipulation, such as Passport, inherits the challenges of both. Instead of failing to compile or throwing a runtime error, bugs in Passport often manifested solely as drops in evaluation numbers. It was challenging to identify whether these drops were caused by bugs to begin with, let alone in which part of the system the bug occurred when there was one.

We are unable to find any work on debugging machine learning systems outside of (potentially very useful) folk knowledge encoded in blog posts[7] and other informal sources. Perhaps a more formal exploration of debugging machine learning systems is warranted. Both better practices [Popel and Bojar 2018] and techniques for improved stability [Liu et al. 2020] may improve the debugging experience. We suspect that improvements to the challenges surrounding the feedback cycle and reproducibility will be not just helpful for but in fact *essential to* improving debugging, as many debugging difficulties are consequences of these challenges.

Other Difficulties. These were only a few of the difficulties we faced as researchers applying machine learning to proof search. These systems are also known to have poor modularity [Sculley et al. 2014] (modifying one component can significantly affect the performance of others); poor explainability [Barredo Arrieta et al. 2020; Gilpin et al. 2018; Guidotti et al. 2018; Lebese et al. 2021] (trained models do not lend themselves to high-level interpretation); and large hardware costs [Heim 2022] (expensive hardware is required to train these models, limiting who can develop them, and often requiring the use of shared clusters which can slow development).

None of these weaknesses are shared by purely symbolic approaches to proof tasks such as proof repair [Ringer et al. 2021], or first-order theorem proving [Czajka and Kaliszyk 2018]. However, current work indicates that tools using these machine learning models can sometimes overcome limitations that current existing purely symbolic tools cannot [First et al. 2020], especially when the solution space is large.

[6] https://github.com/pytorch/pytorch/issues/75240.
[7] http://karpathy.github.io/2019/04/25/recipe/.

	Proverbot-9001	ASTactic	TacTok	Passport
Proof search	✓	✓	✓	✓
Proof state	✓	✓	✓	✓
Tactic history	—	—	✓	✓
Tree-based term encoder	—	✓	✓	✓
Type Constructors	✓	—	—	✓
Global Definitions	✓	—	—	✓
Local Variables	✓	—	—	✓
Paths	—	—	—	✓
Subwords	—	—	—	✓

Fig. 9. A comparison of the features of several proof-synthesis tools.

7 RELATED WORK

We discuss related work in neural proof synthesis, proof corpora, and neural program synthesis.

Neural Proof Synthesis

There have been several other neural proof-synthesis tools for the Coq proof assistant. Figure 9 compares Passport's features to those of prior work. Our work directly enriches the TacTok [First et al. 2020] proof-synthesis tool for Coq (which is, in turn, an enrichment of ASTactic [Yang and Deng 2019]), and evaluates the enriched tool on the CoqGym benchmark suite. TacTok models both proof scripts and proof states to predict tactics. In doing so, however, it erases all tokens from the AST—effectively erasing all syntactic identifier information, including path and file names, local variables, theorem names, type names, and type constructor names. We add these tokens back and explore different design decisions in encoding them, revealing meaningful information about their contributions, and improving over TacTok on the CoqGym benchmark suite. Our insights about syntactic information may provide ideas for dealing with variables used as arguments to tactics in future iterations of TacTok.

Other machine learning tools for Coq include Tactician [Blaauwbroek et al. 2020], Gamepad [Huang et al. 2019], ML4PG [Komendantskaya et al. 2012], and Proverbot9001 [Sanchez-Stern et al. 2020] (which has a web-based frontend, Proofster [Agrawal et al. 2023]). To the best of our knowledge, none of the models in these tools explicitly encode the category a particular identifier belongs to (one of local variable, global definition, or type constructor), none of them encode the path that an identifier comes from, and none of them apply sub-word tokenization. Our insights may help further improve performance of these tools.

We enrich an existing model to explore the impacts of different design decisions for including syntactic information. While the particular architecture of the model we enriched is not the focus of our work, these design decisions may have different impacts depending on the architecture. The model we enriched uses a Tree-LSTM architecture; other models in this space use sequences [Bansal et al. 2019; Blaauwbroek et al. 2020; Sanchez-Stern et al. 2020], other tree architectures [Huang et al. 2019], and graph architectures [Paliwal et al. 2020], with the latter showing significant improvement over previous tree architectures.

Proof-synthesis tools using transformer-based large language models have also begun to emerge [Polu and Sutskever 2020], recently showing promising capabilities for benchmarks in Isabelle/HOL [Jiang et al. 2021; Wu et al. 2022] and Lean [Polu et al. 2023]. These techniques can be used both in a search-based tool [Jiang et al. 2022] (like Passport), and for whole-proof generation [First et al. 2023]. Transformer models that only use the local proof context, such as GPT-f [Polu and Sutskever 2020], cannot derive the identifier information Passport encodes. Capturing that information requires either considering much larger samples of text to process the definitions of each variable (as well as understanding directory structure to derive paths), or running queries against the Coq proof engine, as Passport does. However, future work could enhance transformer models with identifier information, similarly to our approach. Exploring the tradeoffs of different encodings of syntactic information in all of these models may provide interesting insights.

Recent work shows that the decision of whether or not to encode variable names has a significant impact on the performance of a graph neural network for proof synthesis in HOL on the HOList benchmark suite [Paliwal et al. 2020]. Our work explores this tradeoff at a higher level of granularity, looking at the impacts of including different kinds of variables and other syntactic information like paths, and exploring different tokenization decisions and vocabulary sizes. Running a similar experiment on that tool may also prove enlightening.

Proof Corpora

A recent study of proof corpora [Hellendoorn et al. 2018] applying language models found high degrees of naturalness in proofs, and discussed implications for proof engineering tools that could capitalize on that naturalness. The study also found higher degrees of locality than in other programming languages, suggesting that cache-based approaches already helpful in neural program synthesis [Tu et al. 2014] (especially when used in combination with BPE [Karampatsis et al. 2020]) may prove particularly useful for synthesizing proofs. Building a cache on top of BPE is a promising path toward further improving our model performance.

The importance of identifiers is also consistent with recent findings from the REPLica user study of Coq proof engineers [Ringer et al. 2020], which showed a pattern of proof engineers refactoring the names of definitions in predictable and repetitive ways. Furthermore, several of the REPLica benchmarks include syntactic changes in proofs that correspond to semantic changes made alongside them, which points toward syntactic changes possibly revealing useful semantic information that a machine learning tool may be able to pick up on. The REPLica benchmarks may also motivate BPE: One benchmark, for example, shows a change in a type constructor name, along with a change of a substring of the name of a broken lemma that referred to that type constructor name in a way that corresponded to the change. Exploring the performance of Passport on those benchmarks may prove interesting.

Nie et al. [2020a] developed a model for auto-formatting Coq code by encoding spacing information in proof scripts and incorporating techniques from Natural Language Processing. Their work on Roosterize, a toolchain for generation of lemma names [Nie et al. 2020b, 2021], leverages both syntactic and semantic information by combining data from multiple phases of the Coq compiler—tokens, parse trees, and fully elaborated terms. Similar multi-representation approaches may prove an effective means of encoding syntactic information for proof-synthesis models as well.

Specification-mutation analysis can help demonstrate weak specifications, when mutating the definitions does not break the proofs [Celik et al. 2019; Jain et al. 2020]. iCoq [Celik et al. 2017, 2018] and its parallelized version, PiCoq [Palmskog et al. 2018], find failing proof scripts in evolving projects by prioritizing proof scripts affected by a revision. These tools track fine-grained dependencies between Coq definitions, propositions, and proof scripts, to narrow down the potentially affected proof scripts.

Neural Program Synthesis

Neural proof synthesis is similar to neural program synthesis, but adapted to the world of proofs. Neural program synthesis has seen a renaissance of sorts in recent years. The model beneath Github's Copilot code auto-complete tool—Codex—is trained on a large corpus of Github projects, and treats all programs and proofs as text, regardless of the language [Chen et al. 2021]. Another work by DeepMind, AlphaCode, solves a similar task [Li et al. 2022a], as does PaLM-Coder from Google [Chowdhery et al. 2022]. Work at Google [Austin et al. 2021] showed that large language models of this flavor are promising, but struggle to understand the semantics of programs.

A recent YouTube video [Ringer and Cutler 2021] explores the applications of Copilot to proofs, suggesting that even a model trained on raw syntax may suggest helpful hints for small proofs in repetitive files in the CompCert [Leroy 2009] verified C compiler. However, it appears to have limited value for larger, more original proofs with the current data available.

There is a lot we can learn about variable representations and tokenization decisions in neural program synthesis, some of which may be applicable for proofs. Recent work [Tu et al. 2014] shows the benefits of a cache-based model for code completion that exploits locality properties of programs. More recent work [Karampatsis et al. 2020] demonstrates the benefits of BPE tokenization for code completion, especially in combination with cache-based models. Another recent paper [Svyatkovskiy et al. 2021] introduces a framework for evaluating different design decisions for integrating the structure within identifiers within a code completion model, and shows similar benefits for BPE, plus additional benefits from integrating a static analysis to limit the search space. We find similar benefits to BPE in the context of a neural proof-synthesis model, and furthermore show the benefits of tagging different kinds of identifiers and paths differently depending on what kind of information they encode.

Several different models have also been proposed for modeling code, such as AST-like trees [Mou et al. 2014], long-term language models [Dam et al. 2016], and probabilistic grammars [Bielik et al. 2016]. Program synthesis is also widely studied using non-learning based methods, both from types alone [Gvero et al. 2013] and examples and types [Frankle et al. 2016; Osera and Zdancewic 2015].

Identifiers in Code Models

Previous work has been done on providing semantic information for identifiers in code, outside of the context of proof-synthesis. The VarCLR paper explored using contrastive learning to learn which identifiers have similar meanings, in contrast to simply being related [Chen et al. 2022]. It does this by mining variable renamings from GitHub edits, and enables effective use of general purpose language models. Another paper [Karampatsis et al. 2020] explored extensively the tradeoffs of various techniques for dealing with the large vocabulary issues that come from modeling identifiers in code. Several of our design decisions, such as case-sensitivity, and not attempting to split words based on common conventions, are inspired by the results of this article. This article also explores the use of subword tokenizing to handle identifiers in code, and finds it effective. However, their subword architecture is significantly different than ours, since it uses a flat sequence model to model unstructured subword units, while we instead embed a subword model for identifiers inside of a parsed-tree model of the code structure.

8 CONTRIBUTIONS

Our Passport approach enriches a model used for proof synthesis with three different identifier encoding mechanisms: category vocabulary indexing, subword sequence modeling, and path elaboration. We empirically demonstrate that each encoding mechanism improves proof-synthesis performance on the CoqGym benchmark suite. Furthermore, we measured the impact of adding

information for each individual category of identifier: global definitions, local variables, and type constructors. Again, empirically, each category improved performance.

These results are consistent with our intuition that identifiers matter for proofs, that the category of an identifier is useful information, and that drawing connections between identifiers is useful for proof synthesis. Passport automatically proves 12.7% of the theorems in CoqGym, an improvement of 38% over Tok (an example proof-synthesis tool), without changing the core architecture beyond the encoding of identifiers. Combining the new tools developed using the Passport approach with three baseline tools automatically proves 17.2% of the theorems in CoqGym, an improvement of 45% over the baseline tools combined. This intuition and these results will help developers of other tools for program and proof synthesis in other languages beyond Coq, and is a fruitful step toward better tools for engineering robust and reliable formally verified software systems.

APPENDIX

A CATEGORIES OF IDENTIFIERS

Before we implemented Passport, we manually inspected the proof corpora in our training dataset, walking through proofs and analyzing the kinds of information needed to make decisions about which tactic to apply next in a proof. The choice to include identifiers was a product of realizing how much proof engineers rely on naming information to reason about these decisions. But the choice of *which* identifiers to include was less clear. Consider, for example, local variables: Many common local variable names are used in a variety of contexts which may have little relation with one another. A variable named `x` can carry a totally different meaning than the `x` from Figure 3 in Section 3. Without empirical evidence, it was unclear whether an enriched model could potentially suffer performance degradation from drawing fallacious connections like this. As a result, experimental data was an important factor in our selection of which identifiers to include.

Our experiments in Section 5 show that all three categories of identifiers help. In particular, search using the Tok model Passport-enriched with *any one* of the three categories of identifiers alone outperforms search using that model with no identifier information. Furthermore, a search using the Tok model Passport-enriched with *all three* categories of identifiers at once outperforms a search using a Passport-enriched Tok model with just one category of identifiers, for all categories.

The remainder of this Appendix details each of these three categories—global definitions (Appendix A.1), local variables (Appendix A.2), and type constructors (Appendix A.3)—and gives intuition for why each of them may be useful for a tactic prediction model. Finally, Appendix A.4 discusses Passport implementation details.

A.1 Global Definitions

The most straightforward of our categories to include was identifiers referencing global definitions. These identifiers refer to objects defined globally directly by the user, using the keywords `Definition`, `Theorem`, `Inductive`, or one of their variants. Global definitions are generally either an inductive type name, or a name given to some Gallina term (function, constant value, etc.). Crucially, since proof objects themselves are terms, theorems are global definitions with their names bound to their proof objects.

In Coq, most code amounts to creating new global definitions, through a variety of means. The simplest is by writing the term which corresponds to the name explicitly, and using a vernacular command to bind it to the name, as in `Definition n := 5.`. This is commonly how the `Definition` keyword is used, both in defining constant values and in defining functions. When a definition needs to refer to its own name within its body, that is done either using a `fix` in the term, or using the special vernacular keyword `Fixpoint`, which is essentially syntactic sugar for the former.

Global definitions can also be defined interactively, using Coq's tactic system. For example, the proof script in Figure 3 specifies a sequence of tactics which produce a Gallina term referred to by its identifier `posnatMult_comm`. In Gallina, this is indistinguishable from a plain definition—in fact, any term in Coq can be defined using tactics, though this is most common for proofs of lemmas and theorems.

Finally, inductive types can be created using Coq's `Inductive` command. This command creates a new inductive type or type family, given a set of "type constructors," or ways to build objects of the type. When complete, this command defines several objects, including the type itself, its type constructors, and recursion and induction principles for the type. Type constructors are explored in more detail in Appendix A.3.

Encoding the usage of global definitions in terms is extremely useful for predicting tactics. Often, a particular common identifier will signify that certain lemmas will be useful. For instance, in the proof context:

```
n : nat
============================
le (div2 n) n
```

the presence of the `div2` and `le` identifiers indicates that lemmas involving those operators will be useful; in fact, the correct next step is to apply a lemma named `div2_decr`, which applies to goals of the form `le (div2 _)_`. Both `div2` and `le` identifiers correspond to global definitions.

A.2 Local Variables

Besides global definitions, local variables are the most common type of identifier in Coq terms. Local variables can be bound to an explicit term, as in a `let` definition, but in many cases (function parameters, for all bindings, and existential pairs) are given only a type binding. This is in contrast to global definitions, which are always bound directly to terms.

Encoding local variables is often critical to determining the correct next step in a proof, or even understanding its basic structure. Even when the local variable's name is not particularly informative, knowing when local variables repeat is often critical. For example, consider the following proof context (from VST [Appel 2011]):

```
n : nat
============================
n >= div2 n + div2 n
```

If the `n` variable were not the same in all three occurrences, this goal would be impossible to prove without more information. However, because the `n` variable is repeated, this goal holds by the definition of `div2`, which is round-down division by 2.

While local variable names often provide useful information, as mentioned above, common names are often overloaded in their usage. We learned early on that the possibility of performance regression due to uninformative local variables like `x` had concerned the ASTactic authors, and contributed to their decision not to encode identifiers.[8] However, upon closer inspection of the data, we determined that even single-letter identifier names often carry consistent semantic meaning across proofs. The identifier names `hd` and `tl`, for instance, seemed to uniformly refer to the head and tail of a list; because they carried consistent semantic meaning, these identifiers were treated similarly within proofs.

Because of these consistencies in naming, we decided to include local variables.

[8] https://github.com/princeton-vl/CoqGym/discussions/60.

A.3 Type Constructors

Unlike global definitions and local variables, type constructors are not bound on their own, but are instead defined as part of inductive type definitions. As an example of how type constructors are defined, Figure 10 shows the definition of the option type.

```
(* Library Coq, directory Init, file Datatypes.v *)
Inductive option¹ (A² : Type) : Type :=
| Some³ : A → option A
| None³ : option A
```

Fig. 10. The polymorphic `option` datatype in Coq, found in the fully-qualified path `Coq.Init.Datatypes`. Given a type parameter `A`, an `option A` in Coq is one of two things: either it is `Some a` given an element `a` of type `A`, or it is `None`. For consistency, identifiers are highlighted using the same conventions from Figure 2.

The type definition for `option` has two type constructors: `Some`, which creates an `option A` for any object of type `A`, and `None`, which is a constant value of type `option A` for any `A`. There are many examples of such type constructors in common inductive types: `S` and `O` for natural numbers, `cons` and `nil` for lists, and others. Logically, just as type definitions correspond to theorems, type constructors are analogous to introduction rules for types. In the option type in Figure 10, `Some` and `None` encode all possible ways of introducing terms of type `option`. Because of this, type constructors play a special role in deconstructing types—in particular, they appear inside match statements, which *act* on the structure of a type by having one branch per type constructor. Similarly, proofs by induction in Coq *prove* propositions about inductive types by having one case per type constructor.

Knowledge of type constructors can be incredibly useful in determining the next proof step in a proof. In the example from Figure 11, the goal states that `S (S (n + m))` is even, where `m` and `n` are natural numbers. The context shows `(n + m)` is even, but does not include information about `S`. The knowledge that `S` is a successor type constructor of `nat`, and that there exists an `ev` type constructor `ev_SS` of type `ev n -> ev (S (S n))`, is necessary to solve the goal. Here, running the `constructor` tactic results in the goal `ev (n + m)`, which matches one of the hypotheses (IH1).

```
1 subgoal
m, n : nat
E1 : ev n
E2 : ev m
IH1 : ev (n + m)
============================
ev (S (S (n + m)))
```

Fig. 11. A mid-proof context from the first volume of the logical foundations series [Pierce et al. 2021].

A.4 Passport Enrichment Implementation

Enriching the data with these three categories of identifiers amounted to modifying inherited data processing code from TacTok and ASTactic that had erased all information about those identifiers from the data. The inherited code had used the SerAPI [Arias 2016] library to serialize Coq proof objects (terms) as well as proof states and theorems (types), then processed the serialized ASTs returned by SerAPI to erase all identifier information. Enriching the data with two of the three categories of identifiers—definition and local variable names—was a straightforward modification of the post-processing code.

By contrast, adding type constructor names was a more involved process, as Gallina ASTs do not directly store type constructor names. Instead, like its parent type theory, the calculus of inductive constructions [Coquand and Huet 1986; Coquand and Paulin 1990], Coq represents each type constructor in the AST as a tuple consisting of the name of its inductive type together with the index of the particular type constructor.

```
(constructor
  (inductive
    (file_path
      (directory_path [Datatypes; Init; Coq])
      (label option¹)))
  (int 1³))
```

Fig. 12. An unprocessed AST representing a use of the `Some` type constructor for the `option` inductive type from Figure 10, simplified for the sake of presentation. For consistency, identifiers are highlighted using the same conventions from Figure 2, and the index `1` of the `Some` type constructor is highlighted in `yellow`[3]. Note that the identifier of the `Some` type constructor itself is not present.

Figure 12 shows the AST for `Some`, which is the first (type constructors are 1-indexed) type constructor of the `option` datatype. Notably, the AST by default stores the fully-qualified path and name of the inductive type that the type constructor constructs. Thus, the only remaining step is to look up the type constructor from the global environment by passing the fully-qualified name of the inductive type and the index of the type constructor—here, `Coq.Init.Datatypes.option` and `1` —then place it back into the AST where the index is.

To do this, between parsing and encoding, the Passport implementation *unparses* subterms that correspond to type constructor nodes into string representations of the ASTs of the subterms. It then feeds those string representations back through SerAPI, which performs an environment lookup to recover the type constructor name. As with the other identifiers, Passport then inserts a child node containing the identifier into the AST before encoding.

Overall, the Passport approach implementation is 1.5K lines of code and took four developers about a year to build. While the conceptual and design aspects of the Passport approach can extend to all prediction-model-driven, search-based, proof-synthesis tools, the current implementation is straightforwardly applicable to all such tools built within the CoqGym environment [Yang and Deng 2019].

REFERENCES

Agda Development Team. 2007–2021. The Agda Wiki. Retrieved from http://wiki.portal.chalmers.se/agda/pmwiki.php. Accessed 1 August 2022.

Arpan Agrawal, Emily First, Zhanna Kaufman, Tom Reichel, Shizhuo Zhang, Timothy Zhou, Alex Sanchez-Stern, Talia Ringer, and Yuriy Brun. 2023. Proofster: Automated formal verification. In *Proceedings of the Demonstrations Track at the 45th International Conference on Software Engineering* .

Andrew W. Appel. 2011. Verified software toolchain. In *Programming Languages and Systems*. Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, 1–17.

Emilio Jesús Gallego Arias. 2016. SerAPI: Machine-friendly, data-centric serialization for COQ. TechnicalReport. 2016. hal-01384408.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. arXiv:2108.07732. Retrieved from https://arxiv.org/abs/2108.07732.

Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. 2019. HOList: An environment for machine learning of higher-order theorem proving (extended version). *Proceedings of Machine Learning Research (ICML'19, 9-15 June 2019, Long Beach, California, USA)*, Vol. 97.

Alejandro Barredo Arrieta, Natalia Díaz-Rodríguez, Javier Del Ser, Adrien Bennetot, Siham Tabik, Alberto Barbado, Salvador Garcia, Sergio Gil-Lopez, Daniel Molina, Richard Benjamins, Raja Chatila, and Francisco Herrera. 2020. Explainable artificial intelligence (XAI): Concepts, taxonomies, opportunities and challenges toward responsible AI. *Information Fusion* 58 (2020), 82–115. DOI: https://doi.org/10.1016/j.inffus.2019.12.012

Pavol Bielik, Veselin Raychev, and Martin Vechev. 2016. PHOG: Probabilistic model for code. In *Proceedings of the 33rd International Conference on Machine Learning* . Maria Florina Balcan and Kilian Q. Weinberger (Eds.), Proceedings of Machine Learning Research, Vol. 48, PMLR, New York, NY, 2933–2942. Retrieved from http://proceedings.mlr.press/v48/bielik16.html.

Lukas Biewald. 2020. Experiment Tracking with Weights and Biases. Retrieved from https://www.wandb.com/. Accessed 1 August 2022.

Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. 2020. Tactic learning and proving for the Coq proof assistant. In *Proceedings of the International Conference on Logic for Programming, Artificial Intelligence and Reasoning* . Elvira Albert and Laura Kovacs (Eds.), EPiC Series in Computing, Vol. 73, Easy Chair, 138–150. DOI: https://doi.org/10.29007/wg1q

Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2017. ICoq: Regression proof selection for large-scale verification projects. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 171–182. DOI: https://doi.org/10.1109/ASE.2017.8115630

Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2018. A regression proof selection tool for Coq. In *Proceedings of the International Conference on Software Engineering Demonstrations Track*. 117–120. DOI: https://doi.org/10.1145/3183440.3183493

Ahmet Celik, Karl Palmskog, Marinela Parovic, Emilio Jesús Gallego Arias, and Milos Gligoric. 2019. Mutation analysis for Coq. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. 539–551. DOI: https://doi.org/10.1109/ASE.2019.00057

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. arXiv:2107.03374. Retrieved from https://arxiv.org/abs/2107.03374.

Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. 2022. VarCLR: Variable semantic representation pre-training via contrastive learning. In *Proceedings of the 44th International Conference on Software Engineering* . ACM, New York, NY, 2327–2339. DOI: https://doi.org/10.1145/3510003.3510162

Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. 2022. PaLM: Scaling language modeling with pathways. arXiv:2204.02311. Retrieved from https://arxiv.org/abs/2204.02311.

Coq Development Team. 1989–2021. The Coq Proof Assistant. Retrieved from http://coq.inria.fr. Accessed 1 August 2022.

Thierry Coquand and Gérard Huet. 1986. *The Calculus of Constructions*. Technical Report RR-0530. INRIA. Retrieved from https://hal.inria.fr/inria-00076024.

Thierry Coquand and Christine Paulin. 1990. Inductively defined types. In *COLOG-88*. Per Martin-Löf and Grigori Mints (Eds.). Springer Berlin Heidelberg, Berlin, 50–66.

Łukasz Czajka and Cezary Kaliszyk. 2018. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning* 61, 1 (01 Jun 2018), 423–453. DOI: https://doi.org/10.1007/s10817-018-9458-4

Hoa Khanh Dam, Truyen Tran, and Trang Pham. 2016. A deep language model for software code. arXiv:1608.02715. Retrieved from http://arxiv.org/abs/1608.02715.

Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. 2019. Simple high-level code for cryptographic arithmetic—with proofs, without compromises. In *Proceedings of the IEEE Symposium on Security and Privacy*. 1202–1219. DOI: https://doi.org/10.1109/SP.2019.00005

Emily First and Yuriy Brun. 2022. Diversity-driven automated formal verification. In *Proceedings of the 44th International Conference on Software Engineering* . DOI: https://doi.org/10.1145/3510003.3510138

Emily First, Yuriy Brun, and Arjun Guha. 2020. TacTok: Semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages (PACMPL) Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* 4 (November 2020), 231:1–231:31. DOI: https://doi.org/10.1145/3428299

Emily First, Markus Rabe, Talia Ringer, and Baldur Yuriy Brun. 2023. Whole-Proof generation and repair with large language models. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE23)*.

Jonathan Frankle, Peter-Michael Osera, David Walker, and S. Zdancewic. 2016. Example-directed synthesis: A type-theoretic interpretation. *ACM SIGPLAN Notices* 51, 1 (01 2016), 802–815. DOI: https://doi.org/10.1145/2914770.2837629

Philip Gage. 1994. A new algorithm for data compression. *The C Users Journal* 12, 2 (Feb 1994), 23–38.

Xiang Gao. 2022. Cub Device Scan is Not Deterministic as Described in the Documentation #454. Retrieved from https://github.com/NVIDIA/cub/issues/454. Accessed 1 August 2022.

Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael A. Specter, and Lalana Kagal. 2018. Explaining explanations: An approach to evaluating interpretability of machine learning. arXiv:1806.00069. Retrieved from http://arxiv.org/abs/1806.00069.

Riccardo Guidotti, Anna Monreale, Salvatore Ruggieri, Franco Turini, Fosca Giannotti, and Dino Pedreschi. 2018. A survey of methods for explaining black box models. *ACM Computing Surveys* 51, 5, Article 93 (Aug 2018), 42 pages. DOI: https://doi.org/10.1145/3236009

Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* . 27–38. Retrieved from http://infoscience.epfl.ch/record/188990.

Lennart Heim. 2022. Estimating PaLM's Training Cost. Retrieved from https://blog.heim.xyz/author/lennart/. Accessed 1 August 2022.

Vincent J. Hellendoorn, Premkumar T. Devanbu, and Mohammad Amin Alipour. 2018. On the naturalness of proofs. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) New Ideas and Emerging Results Track*. 724–728.

Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. 2019. GamePad: A learning environment for theorem proving. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=r1xwKoR9Y7.

Isabelle Development Team. 1994–2021. Isabelle. Retrieved from http://isabelle.in.tum.de. Accessed 1 August 2022.

Kush Jain, Karl Palmskog, Ahmet Celik, Emilio Jesús Gallego Arias, and Milos Gligoric. 2020. MCoq: Mutation analysis for Coq verification projects. In *Proceedings of the International Conference on Software Engineering Demonstrations Track*. 89–92. DOI: https://doi.org/10.1145/3377812.3382156

Albert Jiang, Konrad Czechowski, Mateja Jamnik, Piotr Milos, Szymon Tworkowski, Wenda Li, and Yuhuai Tony Wu. 2022. Thor: Wielding hammers to integrate language models and automated theorem provers. In *Proceedings of the Conference on Neural Information Processing Systems*.

Albert Qiaochu Jiang, Wenda Li, Jesse Michael Han, and Yuhuai Wu. 2021. LISA: Language models of ISAbelle proofs. In *Proceedings of the Conference on Artificial Intelligence and Theorem Proving*. 17.1–17.3.

Rafael Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code != Big vocabulary: Open-vocabulary models for source code. In *Proceedings of the 42nd International Conference on Software Engineering* . ACM. DOI: https://doi.org/10.1145/3377811.3380342

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* . ACM, New York, NY, 207–220. DOI: https://doi.org/10.1145/1629575.1629596

Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. 2012. Machine learning in proof general: Interfacing interfaces. In *Proceedings of the 10th International Workshop on User Interfaces for Theorem Provers*, Cezary Kaliszyk and Christoph Lüth (Eds.). EPTCS, OPA, Vol. 118, 15–41. DOI: https://doi.org/10.4204/EPTCS.118.2

Lean Development Team. 2014–2021. Theorem Proving in Lean. Retrieved from http://leanprover.github.io/tutorial/. Accessed 1 August 2022.

Thabang Lebese, Ndivhuwo Makondo, Cristina Cornelio, and Naweed Khan. 2021. Proof extraction for logical neural networks. In *Proceedings of the Advances in Programming Languages and Neurosymbolic Systems Workshop*. Retrieved from https://openreview.net/forum?id=Xw3kb6UyA31.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. 2020. GShard: Scaling giant models with conditional computation and automatic sharding. In *Proceedings of the International Conference on Learning Representations*.

Xavier Leroy. 2009. Formal verification of a realistic compiler. *Communications of the ACM* 52, 7 (2009), 107–115. DOI: https://doi.org/10.1145/1538788.1538814

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022a. Competition-Level code generation with AlphaCode. *Science* 378 (2022), 1092–1097. DOI: 10.1126/science.abq1158

Yangguang Li, Feng Liang, Lichen Zhao, Yufeng Cui, Wanli Ouyang, Jing Shao, Fengwei Yu, and Junjie Yan. 2022b. Supervision exists everywhere: A data efficient contrastive language-image pre-training paradigm. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=zq1iJkNk3uN.

Liyuan Liu, Xiaodong Liu, Jianfeng Gao, Weizhu Chen, and Jiawei Han. 2020. Understanding the difficulty of training transformers. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Online, 5747–5763. DOI: https://doi.org/10.18653/v1/2020.emnlp-main.463

Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. 2014. TBCNN: A tree-based convolutional neural network for programming language processing. arXiv:1409.5718. Retrieved from http://arxiv.org/abs/1409.5718.

Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. 2020b. Deep generation of Coq lemma names using elaborated terms. In *Proceedings of the International Joint Conference on Automated Reasoning*. 97–118.

Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. 2020a. Learning to format Coq code using language models. In *Proceedings of the Coq Workshop*.

Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. 2021. Roosterize: Suggesting lemma names for Coq verification projects using deep learning. In *Proceedings of the International Conference on Software Engineering Demonstrations Track*. 21–24. DOI: https://doi.org/10.1109/ICSE-Companion52605.2021.00026

Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices* 50, 6 (June 2015), 619–630. DOI: https://doi.org/10.1145/2813885.2738007

Aditya Paliwal, Sarah Loos, Markus Rabe, Kshitij Bansal, and Christian Szegedy. 2020. Graph representations for higher-order logic and theorem proving. In *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 34, 2967–2974.

Karl Palmskog, Ahmet Celik, and Milos Gligoric. 2018. PiCoq: Parallel regression proving for large-scale verification projects. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*. 344–355. DOI: https://doi.org/10.1145/3213846.3213877

Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. 2018. Deep contextualized word representations. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Vol. 1, Association for Computational Linguistics, New Orleans, LA, 2227–2237. DOI: https://doi.org/10.18653/v1/N18-1202

Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020a. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* . ACM, New York, NY, 771–783. DOI: https://doi.org/10.1145/3324884.3416545

Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020b. Problems and opportunities in training deep learning software systems: An analysis of variance. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* . ACM, New York, NY, 771–783. DOI: https://doi.org/10.1145/3324884.3416545

Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, and Brent Yorgey. 2021. *Software Foundations*. Vol. 1: Logical Foundations. Retrieved from https://softwarefoundations.cis.upenn.edu/lf-current/index.html.

Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. 2023. Formal mathematics statement curriculum learning. *ICLR*. Retrieved from https://arxiv.org/abs/2202.01344.

Stanislas Polu and Ilya Sutskever. 2020. Generative language modeling for automated theorem proving. arXiv:2009.03393. Retrieved from https://arxiv.org/abs/2009.03393.

Martin Popel and Ondřej Bojar. 2018. Training tips for the transformer model. *The Prague Bulletin of Mathematical Linguistics* 110, 1 (2018), 43–70.

Shangshu Qian, Viet Hung Pham, Thibaud Lutellier, Zeou Hu, Jungwon Kim, Lin Tan, Yaoliang Yu, Jiahao Chen, and Sameena Shah. 2021. Are my deep learning systems fair? An empirical study of fixed-seed training. In *Advances in Neural Information Processing Systems*. M. Ranzato, A. Beygelzimer, Y. Dauphin, P. S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34, Curran Associates, Inc., 30211–30227. Retrieved from https://proceedings.neurips.cc/paper/2021/file/fdda6e957f1e5ee2f3b311fe4f145ae1-Paper.pdf.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* . IEEE, Article 20, 16 pages.

Tom P. Reichel. 2022. Large Cumulative Sums Appear to be Nondeterministic. #75240. Retrieved from https://github.com/pytorch/pytorch/issues/75240. Accessed 1 August 2022.

Talia Ringer and Joe Cutler. 2021. Talia and Joe Chat about Proof Engineering with Copilot. Retrieved from https://youtu.be/jFL-ftywPiM. Accessed 1 August 2022.

Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at large: A survey of engineering of formally verified software. *Foundations and Trends®in Programming Languages* 5, 2–3 (2019), 102–281. DOI: https://doi.org/10.1561/2500000045

Talia Ringer, RanDair Porter, Nathaniel Yazdani, John Leo, and Dan Grossman. 2021. Proof repair across type equivalences. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM. DOI: https://doi.org/10.1145/3453483.3454033

Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. 2020. REPLica: REPL instrumentation for Coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs* . ACM, New York, NY, 99–113. DOI: https://doi.org/10.1145/3372885.3373823

Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner. 2020. Generating correctness proofs with neural networks. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. ACM SIGPLAN.

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. 2014. Machine learning: The high interest credit card of technical debt. In *Proceedings of the NIPS 2014 Workshop on Software Engineering for Machine Learning*.

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural machine translation of rare words with subword units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, 1715–1725. DOI: https://doi.org/10.18653/v1/P16-1162

Gil Shamir and Dong Lin. 2022. Reproducibility in Deep Learning and Smooth Activations. Retrieved from https://ai.googleblog.com/2022/04/reproducibility-in-deep-learning-and.html?m=1. Accessed 1 August 2022.

Jean Souyris. 2014. Industrial Use of CompCert on a Safety-Critical Software Product. Retrieved from http://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf. Accessed 1 August 2022.

Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. *MSR* 18 (2021), 329–340. https://doi.org/10.1109/MSR52588.2021.00045

Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Vol. 1, 1556–1566. DOI: https://doi.org/10.3115/v1/P15-1150

Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the localness of software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* . ACM, New York, NY, 269–280. DOI: https://doi.org/10.1145/2635868.2635875

James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* . ACM, New York, NY, 357–368. DOI: https://doi.org/10.1145/2737924.2737958

Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. 2022. Memorizing transformers. In *Proceedings of the International Conference on Learning Representations*. Retrieved from https://openreview.net/forum?id=TrjbxzRcnf-.

Kaiyu Yang and Jia Deng. 2019. Learning to prove theorems via interacting with proof assistants. In *Proceedings of the International Conference on Machine Learning*. Retrieved from http://proceedings.mlr.press/v97/yang19a/yang19a.pdf.

Received 1 August 2022; revised 1 February 2023; accepted 24 March 2023