UNIVERSITY OF CALIFORNIA SAN DIEGO

Hybrid-Neural Synthesis of Machine-Checkable Software Correctnesss Proofs

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science

by

Alex Sanchez-Stern

Committee in charge:

Professor Sorin Lerner, Chair
Professor Samuel Buss
Professor Todd Millstein
Professor Nadia Polikarpova
Professor Lawrence Saul

2021

The dissertation of Alex Sanchez-Stern is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

DEDICATION

To all the inspiring graduate students and faculty I met along my journey, and to my late grandfather, Professor Edward Stern.

EPIGRAPH

As long as I'm learning something, I figure I'm okay -

it's a decent day.

—Hunter S. Thompson

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

Thanks to my friends and family, without which I would have given up long ago. Thanks to Zach Tatlock and Pavel Panchekha for starting me down the path of research, Joseph Redmon for starting me down the path of this line of research, and Sorin Lerner for guiding me steadfastly. And thanks to all the graduate students who came before me and showed me the way.

The work described in this thesis was done in conjunction with Yousef Alhessi, Valentin Robert, Joe Redmon, Sicun Gao, Lawrence Saul, and Sorin Lerner.

Section 3.3, Chapter 4, Chapter 6, and Section 7.1 are rewritten and updated versions of material published at Machine Learning for Programming Languages (MAPL) 2020. Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner 2020. The dissertation author was the primary author of this paper.

Chapter 5 and Section 7.2 is based on ongoing work performed by Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, Sicun Gao, and Sorin Lerner. The dissertation author is the primary author of this work.

VITA

2015	B. S. in Computer Science *with honors*, University of Washington, Seattle
2016	M. S. in Computer Science, University of Washington, Seattle
2021	PhD in Computer Science, UC San Diego, San Diego

PUBLICATIONS

Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, Sorin Lerner, "Generating Correctness Proofs with Neural Networks", *Machine Learning for Programming Languages*, 2020

Alex Sanchez-Stern, Pavel Panchekha, Sorin Lerner, Zachary Tatlock, "Finding Root Causes of Floating Point Error", *Programming Languages Design and Implementation*, 2018

John Renner, Alex Sanchez-Stern, Fraser Brown, Sorin Lerner, Deian Stefan, "Scooter & Sidecar: A Domain-Specific Approach to Writing Secure Migrations", *Programming Languages Design and Implementation*, 2021

Talia Ringer, Alex Sanchez-Stern, Dan Grossman, Sorin Lerner, "REPLica: REPL instrumentation for Coq Analysis", *Certified Proofs and Programs*, 2020

Nasrine Damouche, Matthieu Martel, Pavel Panchekha, Jason Qiu, Alex Sanchez-Stern, Zachary Tatlock, "Towards a Standard Benchmark Format and Suite for Floating-Point Analysis", *Numerical Software Verification*, 2016

Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, Zachary Tatlock, "Automatically Improving Accuracy for Floating Point Expressions", *Programming Languages Design and Implementation, Distinguished Paper*, 2015

ABSTRACT OF THE DISSERTATION

Hybrid-Neural Synthesis of Machine-Checkable Software Correctnesss Proofs

by

Alex Sanchez-Stern

Doctor of Philosophy in Computer Science

University of California San Diego, 2021

Professor Sorin Lerner, Chair

Foundational verification allows programmers to build software which has been empirically shown to have high levels of assurance in a variety of important domains. However, the cost of producing foundationally verified software remains prohibitively high for most projects, as it requires significant manual effort by highly trained experts. In my thesis, I developed Proverbot9001, a proof search system using machine learning techniques to produce proofs of software correctness in interactive theorem provers. Proverbot9001 is composed of three parts: a supervised predictor model, a reinforcement predictor model, and a search system which can make use of either model to automatically prove theorems.

I demonstrated Proverbot9001's supervised model on the proof obligations from a large

practical proof project, the CompCert verified C compiler, and show that it can effectively automate what were previously manual proofs, automatically producing proofs for 36% of theorem statements in our test dataset, when combined with solver-based tooling. Without any additional solvers, we exhibit a proof completion rate that is a 4X improvement over prior state-of-the-art machine learning models for generating proofs in Coq. I then demonstrated that for some classes of proofs, the reinforcement model can build upon the supervised model and produce even more proofs with no additional labeled-data.

Chapter 1

Introduction

A promising approach to software verification is *foundational verification*. In this approach, programmers use an interactive theorem prover, such as Coq [FHB+20] or Isabelle/HOL [Pau93], to state and prove properties about their programs. The proofs are performed interactively via the use of *proof commands*, which programmers invoke to make progress on a proof. To complete a proof, a programmer must provide guidance to the proof assistant at each step by picking which proof command to apply. Foundational verification has shown increasing promise over the past two decades; it has been used to prove properties of programs in a variety of settings, including compilers [Ler09], operating systems [KEH+09], database systems [MMSW10], file systems [CCK+17], distributed systems [WWP+15], and cryptographic primitives [App15].

One of the main benefits of foundational verification is that it provides high levels of assurance. The interactive theorem prover makes sure that proofs of program properties are done in full and complete detail, without any implicit assumptions or forgotten proof obligations. Furthermore, once a proof is completed, foundational proof assistants can generate a representation of the proof in a foundational logic; these proofs can be checked with a small kernel. In this setting only the kernel needs to be trusted (as opposed to the entire proof assistant), leading to a

small trusted computing base. As an example of this high-level of assurance, a study of compiler bugs [YCER11] has shown that CompCert [Ler09], a compiler proved correct in the Coq proof assistant, is significantly more robust than its non-verified counterparts.

Unfortunately, the benefits of foundational verification come at a great cost. The process of performing proofs in a proof assistant is extremely laborious. CompCert [Ler09] took 6 person-years and 100,000 lines of Coq to write and verify, and seL4 [KEH+09], which is a verified version of a 10,000 line operating system, took 22 person-years to verify. The sort of manual effort is one of the main impediments to the broader adoption of proof assistants and foundational verification.

In this thesis, I developed Proverbot9001, a novel system that uses machine learning to help alleviate the manual effort required to complete proofs in an interactive theorem prover. The source of Proverbot9001 is publicly available on GitHub [1].

1.1 This Work

Proverbot9001 is composed of three main systems:

1. A tactic prediction model, trained using supervised learning
2. A tactic evaluation model, trained using reinforcement learning
3. A proof search system which can make use of the previous models to produce proofs for a given lemma statement

Altogether, Proverbot9001 trains on existing proofs to learn a prediction model, interacts with the proof environment to refine that model, and then uses that model to guide proof search and complete proofs.

[1] https://github.com/UCSD-PL/proverbot9001

Supervised Tactic Prediction The main contribution of this system is bringing domain knowledge to the feature engineering and model architecture of learned tactic predictors. In particular, our work distinguishes itself from prior work on machine learning for proofs in two ways:

1. A two part tactic-prediction model, in which prediction of tactic arguments is primary and informs prediction of tactics themselves.

2. An argument prediction architecture which makes use of recurrent neural networks over sequential representations of terms.

Reinforced Tactic Evaluation Our reinforced tactic evaluation improves on the state-of-the-art for completing proofs within some domains. It's contributions include:

1. A method of combining a state evaluator network with our tactic prediction network to improve predictions, sharing several learned components.

2. A system for training the state evaluation network using Q value lookahead in the context of Coq proofs.

Guided Proof Search Our guided proof search system is able to make effective use of either the bare supervised predictor, or the combined reinforced predictor, to effectively search for and find Coq proofs.

It improves over general tree search and previous work with:

1. Several effective tree pruning techniques inside of a prediction-guided proof search.

2. Multiple depth bounding techniques adapted to the proof search problem.

Results We tested Proverbot9001 end-to-end by training on the proofs from 162 files from CompCert, and testing on the proofs from 13 files[2]. When combined with solver-based tooling

[2]This training/test split comes from splitting the dataset 90/10, and then removing from the test set files that don't contain proofs.

(which alone can only solve 7% of proofs), Proverbot9001 can automatically produce proofs for 36% of the theorem statements in our test dataset (182/501). In our default configuration without external solvers, Proverbot9001 solves (produces a checkable proof for) 19.36% (97/501) of the proofs in our test set, which is a nearly 4X improvement over the previous state of the art system that attempts the same task [YD19]. Our model is able to reproduce the tactic name from the solution 32% of the time; and when the tactic name is correct, our model is able to predict the solution argument 89% of the time. We also show that Proverbot9001 can be trained on one project and then effectively predict on another project.

1.2 Outline of Dissertation

The rest of this dissertation is organized as follows. First, in Chapter 2, we'll give an overview of some of the background needed to understand the rest of the work, including foundational verification and machine learning. Next, in Chapter 3, we'll talk about the overall function of Proverbot9001, and how it's pieces will fit together, as well as some definitions that will be used in the rest of the dissertation. Then, Chapter 4, Chapter 5, and Chapter 6, we will discuss the main systems of Proverbot9001 in detail. Finally, we'll conclude with Chapter 8 and Chapter 9.

Chapter 2

Background

2.1 Foundational Verification

Program verification is a well studied problem in the programming languages community. Most work in this field falls into one of two categories: solver-backed automated (or semi-automated) techniques, where a simple proof is checked by a complex procedure; and foundational logic based techniques, where a complex proof is checked by a simple procedure.

While research into solver-backed techniques has produced fully-automated tools in many domains, these approaches are generally incomplete, failing to prove some desirable propositions. When these procedures fail, it is often difficult or impossible for a user to complete the proof, requiring a deep knowledge of the automation. In contrast, foundational verification techniques require a heavy initial proof burden, but scale to any proposition without requiring a change in proof technique. However, the proof burden of foundational techniques can be prohibitive; CompCert, a large and well-known foundationally verified compiler, took 6 person-years of work to verify [KBW+18], with other large verification projects sporting similar proof burdens.

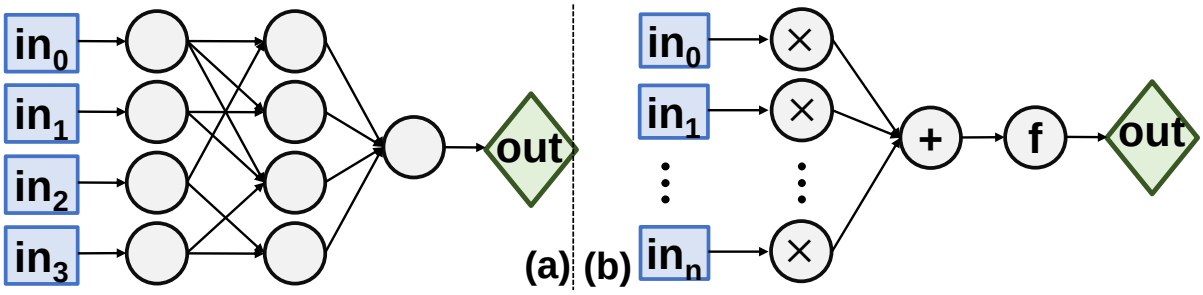


Figure 2.1: (a) A feed-forward neural network, where each individual gray circle is a perceptron (b) An individual perceptron, which multiplies all the inputs by weights, sums up the results, and then applies a non-linear function f.

2.2 Interactive Theorem Provers

Most foundational (and some solver-backed) verification is done in an *interactive* theorem prover. Interactive theorem provers allow the user to define proof goals alongside data and program definitions, and then prove those goals interactively, by entering commands which manipulate the proof context. The name and nature of these commands varies by the proof assistant, but in many foundational assistants, these commands are called "tactics", and coorespond to primitive proof techniques like "induction", as well as search procedures like "omega" (which searches for proofs over ring-like structures). Proof obligations in such proof assistants take the form of a set of hypotheses (in a Curry-Howard compatible proof theory, bound variables in a context), and a goal (a target type); proof contexts may consist of multiple proof obligations.

2.3 Supervised Learning

Machine learning is an area of computer science dating back to the 1950s. In problems of supervised learning, the goal is to learn a function from labeled examples of input-output pairs. Models for supervised learning parameterize a function from inputs to outputs and have a procedure to update the parameters from a data set of labeled examples. Machine learning has traditionally been applied to problems such as handwriting recognition, natural language

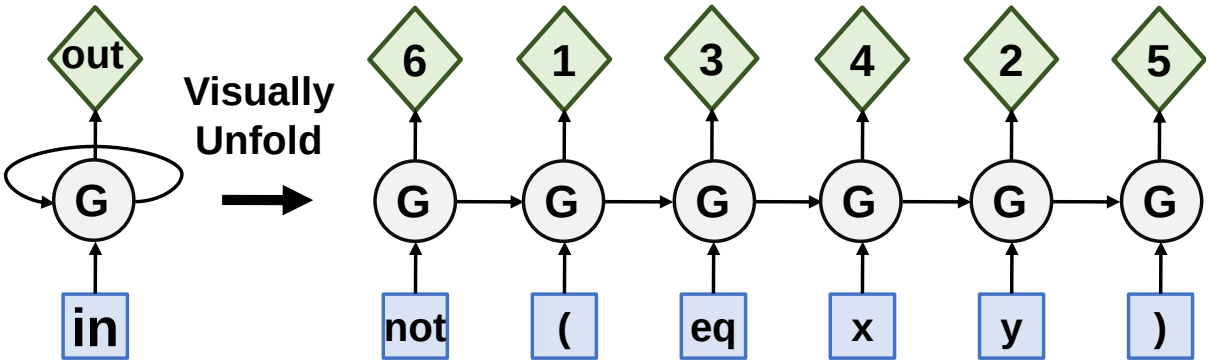


Figure 2.2: A recurrent neural network. Inputs are in blue boxes at the bottom, and each iteration produces an output value, as well as a new state value for the next iteration.

processing, and recommendation systems.

2.4 Reinforcement Learning

Reinforcement learning is a form of semi-supervised learning that acts within an environment to learn a policy of action. While supervised learning, in it's simplest form, models a single function from inputs to outputs, reinforcement learning necessarily includes a broader model of how outputs will explore a space. Reinforcement learning has been applied successfully to many control problems, including learning to play video games [MKS+13], control robots [KBP13], and beat human players at Go [SSS+17].

A reinforcement model has two parts: a set of states, and a set of actions (see example in Figure 2.3). You can think of the state space as a connected graph, where each state is a node in the graph, and the actions are its edges. Each edge also has a **reward**, some positive or negative value. An actor in this space can have a **policy**, a function which gives an action to each state, and can be applied iteratively to move about the state space. The goal of a reinforcement learning algorithm is to find a policy which, when followed, maximizes the reward encountered by the agent.

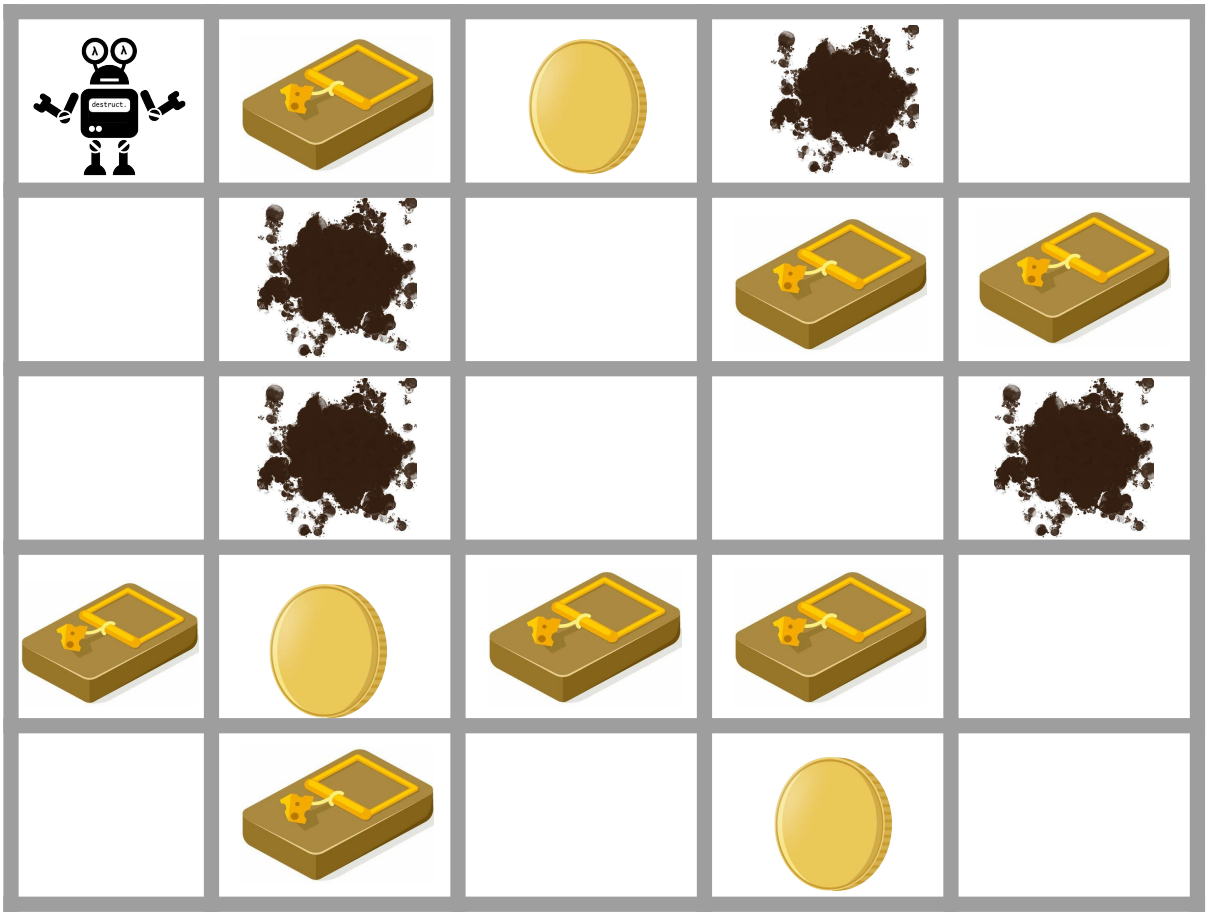


Figure 2.3: An example of an environment for reinforcement learning. Here the states are arranged in a grid, with edges between adjacent states. Moving onto a square with a coin nets a positive reward, while moving to a square with a mousetrap nets a large negative reward. Moving into a square with mud gives only a small negative reward, and all other squares have no reward.

2.5 Neural Networks

Neural Networks are a particular class of learned model where layers of nodes are connected together by a linear combination and a non-linear activation function, to form general function approximators. Neural Networks have a variety of structures, some forming a straightforward "stack" of nodes with some connections removed (convolutional), and others, such as those used for natural language processing, using more complex structures like loops.

We will make use of two different kinds of neural networks: feed-forward networks and recurrent neural networks. Figure 2.1(a) shows the structure of a feed-forward network, where each gray circle is a perceptron, and Figure 2.1(b) shows individual structure of a perceptron.

Figure 2.2 shows the structure of a recurrent neural network (RNN). Inputs are shown in blue, outputs in green and computational nodes in gray. The computational nodes are Gated Recurrent Network nodes, GRU for short, a commonly used network component with two inputs and two outputs [CvMG+14]. The network is recurrent because it feeds back into itself, with the state output from the previous iteration feeding into the state input of the next iteration. When we display an RNN receiving data, we visually unfold the RNN, as shown on the right side of Figure 2.2, even though in practice there is still only one GRU node. The right side of Figure 2.2 shows an example RNN that processes tokens of a Coq goal, and produces some output values.

Chapter 3

Overview

3.1 Usage and Infrastructure

From a usage perspective, Proverbot9001 is made of several command-line utilities corresponding to various aspects of the scraping, training, and searching process.

If a user wants to start using Proverbot9001 right away, they can download pre-trained weights from the CompCert project using a Makefile target, and immediately start searching for solutions to the lemmas they care about. Our experiments have suggested that Proverbot9001 performs reasonably well even when run on projects that are seperate from the one it was trained on (see Section 7.1.4).

However, Proverbot9001 can perform even better when trained on existing proofs from the project it will be used for, or other projects with a similar proof style. In that case, the user can "scrape" any proofs from a compatible Coq version (8.9-8.12) to produce training files, and train the model on these training files.

Scraping proof files The first a user must do to train Proverbot9001 on a new set of proofs is "scrape" the proof data. Scraping takes a set of `*.v` files, and a project root, and produces a scrape file containin all the information needed to reconstruct the running of the file. Scrape

files are in a simple JSON-based file format, where each line is a single object detailing a proof interaction. This ensures that scrapes for multiple projects can be easily combined, say, for combining a small number of existing proofs from your project with a larger number of proofs from a similar project, to train a model to complete your project.

Users can scrape a proof file or set of proof files with the command:

```
python3 $PROVERBOT_ROOT/src/scrape.py --prelude=$PROJECT_ROOT \
  projfile1.v projfile2.v ...
```

Because the scraping process involves running the file through Coq, the project root must be provided so that it can find the project configuration (in a file called `_CoqProject`. Many other optional flags can be passed to the scraper, controlling its output location, number of threads used, failure modes, and verbosity.

By default, the scraper will attempt to linearize all proofs in the source files before scraping them (see Section 4.2.2). This can be disabled with a flag. In the course of developing the scraper, we developed our own Python API to Coq, available on Github [1].

Training the supervised model Once we have a scrape of the interactions in the file, we can use those interactions to train our supervised model.

However, not all proof interactions can be learned from by our model. In particular, commands that include unlinearized semicolons, or commands whose arguments do not fit Proverbot9001s argument model, cannot be used as training data. Additionally, users may want to try training the model only on certain types of tactics, or certain types of arguments.

For that purpose, we have designed a *interaction filter DSL*, a small language which encodes boolean propositions over proof interactions. Programs in this language are passed to the model training command, and run on each proof interaction to determine whether it should be included in the training data or not. Figure 3.1 shows the syntax and basic semantics of this

[1] https://github.com/HazardousPeach/coq_serapy

$$
\begin{array}{lrl}
\text{top-expr} & E & ::= e_1 + e_2 + ... \mid e_1 \% e_2 \% ... \\
\text{expr} & e & ::= (E) \mid atom \\
\text{atom} & atom & ::= \text{none} \mid \text{all} \\
 & & \mid \text{goal-args} \mid \text{hyp-args} \\
 & & \mid \text{rel-lemma-args} \mid \text{numeric-args} \\
 & & \mid \text{no-semis} \mid \text{default} \\
 & & \mid \text{tactic:}\textit{string} \mid \text{etactic:}\textit{string} \\
 & & \mid \text{maxargs:}\textit{nat}
\end{array}
$$

Figure 3.1: The syntax of interaction filter programs in Proverbot9001. `+` evaluates to `or` and `\%` evaluates to `and`. The atoms "none" and "all" are the constant functions for False and True respectively; "goal-args", "hyp-args", "rel-lemma-args", and "numeric-args" control what kinds of arguments will be accepted; "no-semis" and "default" control the handling of punctuation and vernacular commands; "tactic" only returns true for tactics that match a string, while "etactic" accepts the "e"-prefixed version of that tactic too; and "maxargs" controls the number of arguments each tactic can have (for our models, 1).

context filter language. The Makefile for Proverbot9001 includes a context filter program to match the argument model of our predictor.

Users can train a model on a scrape file using the command:

```
python3 $PROVERBOT_ROOT/src/proverbot9001.py train polyarg \
  --context-filter=$FILTER $SCRAPE_FILE $OUT_WEIGHTS
```

The training command also accepts a number of parameters, controlling the hyperparameters of the model, as well as enabling and disabling certain subsystems, and controlling how metadata is handled.

Refining the model with reinforcement Once you have a set of supervised weights, you can refine them using reinforcement learning to produce a set of reinforced weights. To do so, you must provide the supervised weights, as well as a set of proof files to be used as *environments* for interaction. Not all proofs are equally suitable as a reinforcement learning environment; therefore users can also pass a file containing a list of lemma names, and only those lemmas from the proof files will be used for reinforcement. By default, all proofs in the provided proof files are used for reinforcement.

The supervised learning will also pre-populate the experience buffer with a given scrape file, and potentially use that scrape file for pretraining (see Section 5.3.2).

Users can produce a set of reinforced weights using the command:

```
python3 $PROVERBOT_ROOT/src/reinforce.py \
  --predictor-weights=$SUPERVISED_WEIGHTS \
  $INITIAL_SCRAPE $OUT_WEIGHTS enffile1.v envfile2.v ...
```

This command also takes a variety of arguments to control exploration, training, and the use of demonstrations (see Section 5.3.4).

Once a set of evaluator weights have been trained, you can combine them with the supervised weights to produce *reinforced predictor* weights. Users produce these weights using the command:

```
python3 $PROVERBOT_ROOT/src/mk_reinforced_weights.py \
  $SUPERVISED_WEIGHTS $EVALUATOR_WEIGHTS $OUT_WEIGHTS
```

Searching for proofs Finally, you can invoke the search system to use either a bare supervised predictor, or a reinforced predictor, to produce the solutions to proofs.

To set up your proof files for search, make sure they run without modification in your projects configuration. Then, write and admit each of the lemmas that you would like to solve in your proof file, ensuring that all relevant definitions are imported.

Then, invoke the command:

```
python3 $PROVERBOT_ROOT/src/search_file.py --weightsfile=$WEIGHTS \
   problem_file_1.v problem_file_2.v ...
```

Be mindful of the order in which you state your lemmas; when solving a particular lemma, Proverbot9001 will attempt to use all lemmas defined earlier in the file. If there are particular lemmas that you would like Proverbot9001 to attempt to use in the solutions, you can give their definitions earlier in the file. Alternatively, if those lemmas definitions are in an imported library or file, you can put their names in a file, one per line, and pass that file with the flag `--add-env-lemmas`.

You can also add new axioms that are not present in dependencies by puttin gtheir full definitions in a file, one per line, and passing it in with the `--add-axioms` command. However, these axioms must parse and typecheck before every proof that Proverbot9001 attempts, and the proofs produced will not on their own check, without extra insertion of the axiom.

3.2 Process

In this section, we'll present Proverbot9001's prediction and search process at a high level with an example from CompCert. You can see the top-level structure of Proverbot9001 in Figure 3.2.

Consider the following theorem from the CompCert compiler:

```
Definition binary_constructor_sound
    (cstr: expr -> expr -> expr)
    (sem: val -> val -> val) : Prop :=
  forall le a x b y,
  eval_expr ge sp e m le a x ->
  eval_expr ge sp e m le b y ->
  exists v, eval_expr ge sp e m le (cstr a b) v
            /\ Val.lessdef (sem x y) v.

Theorem eval_mulhs:
  binary_constructor_sound mulhs Val.mulhs.
Proof.
...
```

This theorem states that the `mulhs` expression constructor is sound with respect to the specification `Val.mulhs`.

At the beginning of the proof of `eval_mulhs`, Proverbot9001 predicts three candidate

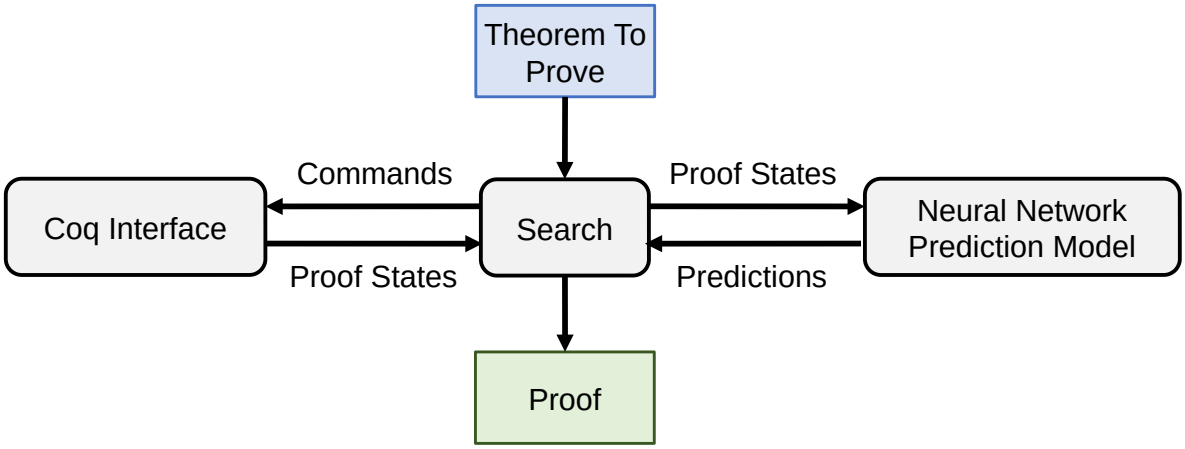


Figure 3.2: The overall architecture of Proverbot9001, built using CoqSerapi, Python, and PyTorch.

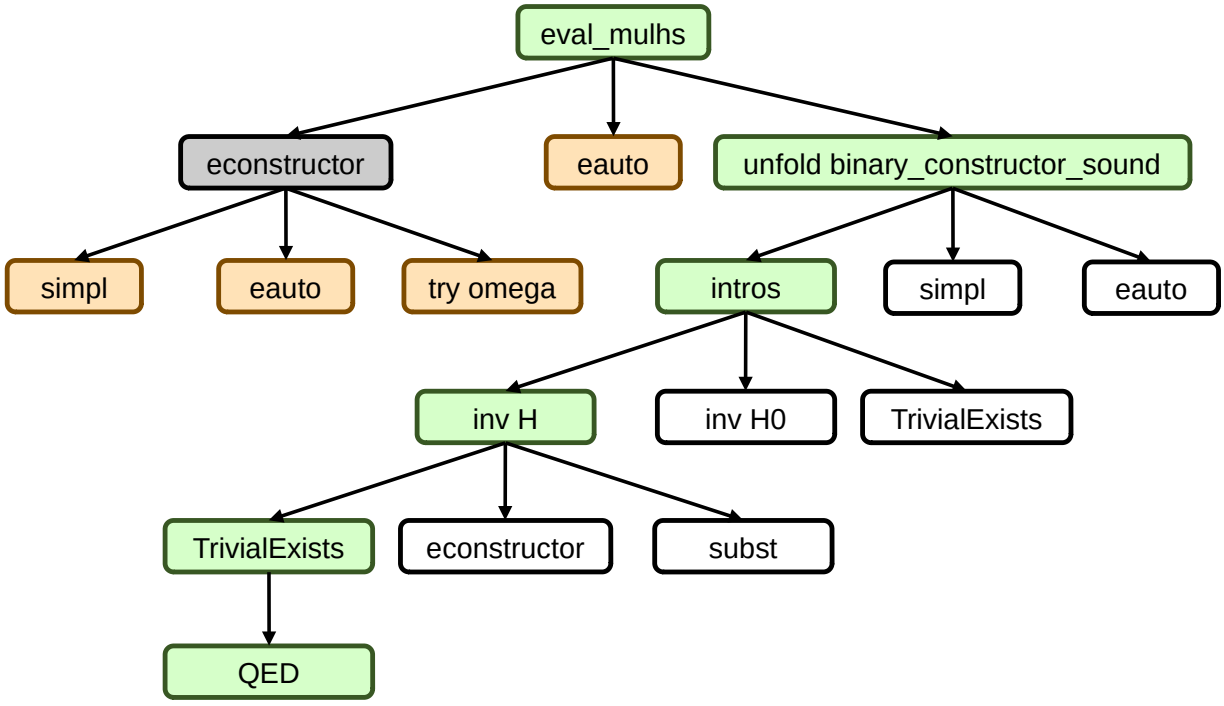


Figure 3.3: A graph of a Proverbot9001 search. In green are the tactics that formed part of the discovered solution, as well as the lemma name and the QED. In orange are nodes that resulted in a context that is at least as hard as one previously found (see Chapter 6).

$\mathcal{T}$	Tactics
$\mathcal{A}$	Tactic arguments
$\mathcal{C} = \mathcal{T} \times \mathcal{A}$	Proof commands
I	Identifiers
$\mathcal{Q}$	Propositions
$\mathcal{G} = \mathcal{Q}$	Goals
$\mathcal{H} = I \times \mathcal{Q}$	Hypotheses
$\mathcal{O} = [\mathcal{H}] \times \mathcal{G}$	Obligations
$\mathcal{S} = [\mathcal{O} \times [\mathcal{C}]]$	Proof states

Figure 3.4: Formalism to model a Proof Assistant

tactics, `econstructor`, `eauto`, and `unfold binary_constructor_sound`. Once these predictions are made, Proverbot9001 tries running all three, which results in three new states of the proof assistant. In each of these three states, Proverbot9001 again makes predictions for what the most likely tactics are to apply next. These repeated predictions create a search tree, which Proverbot9001 explores in a depth first way. The proof command predictions that Proverbot9001 makes are ordered by likelihood, and the search explores more likely branches first.

Figure 3.3 shows the resulting search tree for `eval_mulhs`. The nodes in green are the nodes that produce the final proof. Orange nodes are predictions that fail to make progress on the proof (see Chapter 6); these nodes are not expanded further. All the white nodes to the right of the green path are not explored, because the proof in the green path is found first.

3.3 Definitions

In the rest of the paper, we will describe the details of how our predictors work. We start with a set of definitions that will be used throughout. In particular, Figure 3.4 shows the formalism we will use to represent the state of an in-progress proof. A tactic $\tau \in \mathcal{T}$ is a tactic name. An argument $a \in \mathcal{A}$ is a tactic argument. For simplicity of the formalism, we assume that all tactics take zero or one arguments. We use I for the set of Coq identifiers, and Q for the set of Coq propositions. A *proof state* $\sigma \in \mathcal{S}$ is a state of the proof assistant, which consists of a

list of obligations along with their proof command history. We use $[X]$ to denote the set of lists of elements from X. An obligation is a pair of: (1) a set of hypotheses (2) a goal to prove. A hypothesis is a proposition named by an identifier, and a goal is a proposition.

Section 3.3 is a rewritten version of material published at at Machine Learning for Programming Languages (MAPL) 2020. Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner 2020. The dissertation author was the primary author of this paper.

Chapter 4

Supervised Learning

4.1 Model

We start by explaining how we predict individual steps in the proof. Once we have done this, we will explain how we use these proof command predictions to guide a proof search procedure.

We define $\mathcal{D}[\tau]$ to be a scoring function over τ, where larger scores are preferred over smaller ones:

$$\mathcal{D}[\tau] = \tau \to \mathbb{R}$$

We define a τ-predictor $\mathcal{R}[\tau]$ to be a function that takes a proof state $\sigma \in \mathcal{S}$ (*i.e.* a state of the proof assistant under which we want to make a prediction) and returns a scoring function over τ. In particular, we have:

$$\mathcal{R}[\tau] = \mathcal{S} \to \mathcal{D}[\tau]$$

Our main predictor P will be a predictor of the next step in the proof, *i.e.* a predictor for proof commands:

$$P : \mathcal{R}[\mathcal{T} \times \mathcal{A}]$$

We divide our main predictor into two predictors, one for tactics, and one for arguments:

$$P_{tac} : \mathcal{R}[\mathcal{T}]$$

$$P_{arg} : \mathcal{T} \to \mathcal{R}[\mathcal{A}]$$

Our main predictor P combines P_{tac} and P_{arg} as follows:

$$P(\sigma) = \lambda(\tau, a)\ .\ P_{tac}(\sigma)(\tau)\ \otimes\ P_{arg}(\tau)(\sigma)(a)$$

where $\otimes$ is an operator that combines the scores of the tactic and the argument predictors. We now describe the three parts of this prediction architecture in turn: P_{tac}, P_{arg}, and $\otimes$.

4.1.1 Predicting Tactics (P_{tac})

To predict tactics, Proverbot9001 uses of a set of manually engineered features to reflect important aspects of proof prediction: (1) the head of the goal as an integer (2) the name of the previously run tactic as an integer (3) a hypothesis that is heuristically chosen (based on string similarity to goal) as being the most relevant to the goal (4) the similarity score of this most relevant hypothesis.

These features are embedded into a continuous vector of 128 floats using a standard word embedding, and then fed into a fully connected feed-forward neural network (3 layers, 128 nodes-wide) with a softmax (normalizing) layer at the end, to compute a probability distribution over possible tactic names. This architecture is trained on 153402 samples with a stochastic gradient descent optimizer.

The architecture of this model is shown in Figure 4.1. Blue boxes represent input; purple boxes represent intermediate encoded values; green boxes represent outputs; and gray circles represent computations. The NN circle is the feed-forward Neural Network mentioned above.

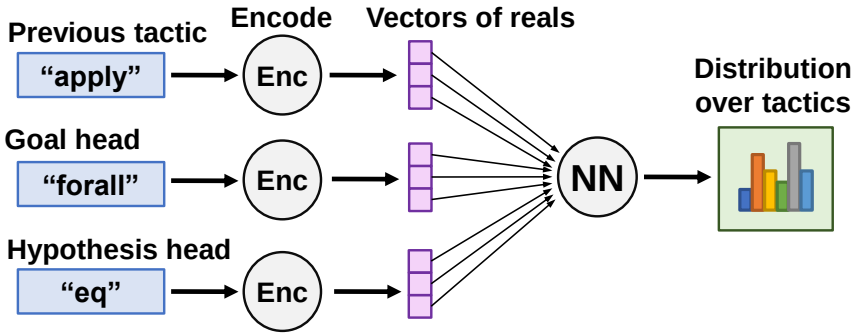


Figure 4.1: Proverbot9001's model for predicting tactics. Takes as input three features for each data point: the previous tactic run, the head token of the goal, and of the most relevant hypothesis (see Section 4.1.1). We restrict the previous tactic feature to the 50 most common tactics, and head tokens on goal and hypothesis to the 100 most common head tokens.

The Enc circle is a word embedding module.

4.1.2 Predicting Tactic Arguments (P_{arg})

Once a tactic is predicted, Proverbot9001 next predicts arguments. Recall that the argument predictor is a function $P_{arg} : \mathcal{R}[\mathcal{A}]$. In contrast to previous work, our argument model is a prediction architecture in its own right.

Proverbot9001 currently predicts zero or one tactic arguments; However, since the most often-used multi-argument Coq tactics can be desugared to sequences of single argument tactics (for example "`unfold a, b`" to "`unfold a. unfold b.`"), this limitation does not significantly restrict our expressivity in practice.

Proverbot9001 makes three kinds of predictions for arguments: *goal-token* arguments, *hypothesis* arguments, *lemma* arguments:

Goal-token arguments are arguments that are a single token in the goal; for instance, if the goal is `not (eq x y)`, we might predict `unfold not`, where `not` refers to the first token in the goal. In the case of tactics like `unfold` and `destruct`, the argument is often (though not always) a token in the goal.

Hypothesis arguments are identifiers referring to a hypothesis in context. For instance, if

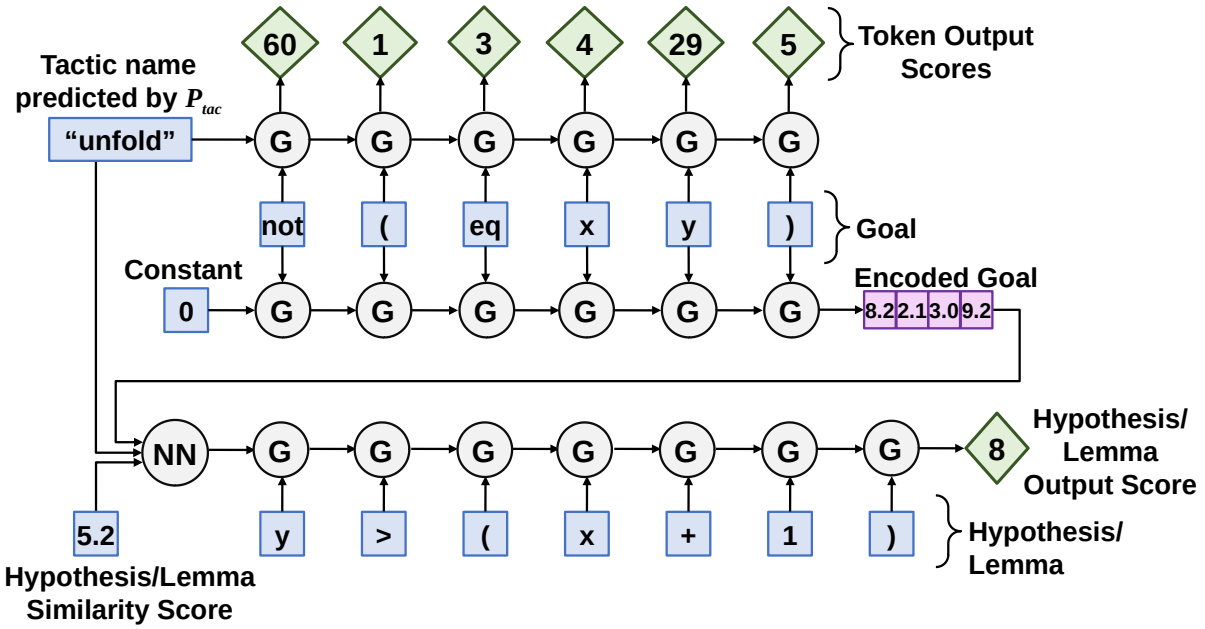


Figure 4.2: The model for scoring possible arguments.

we have a hypothesis `H` in context, with type `is_path (cons (pair s d) m)`, we might predict `inversion H`, where `H` refers to the hypothesis, and `inversion` breaks it down. In the case of tactics like `inversion` and `destruct`, the argument is often a hypothesis identifier.

Finally, *lemma* arguments are identifiers referring to a previously defined proof. These can be basic facts in the standard library, like

```
plus_n_0 : forall n : nat, n = n + 0
```

or a lemma from the current project, such as the `eval_mulhs` described in the overview. In Proverbot9001, lemmas are considered from a subset of the possible lemma arguments available in the global context, in order to make training tractable. Proverbot9001 supports several different modes for determining this subset; by default we consider lemmas defined previously in the current file.

The architecture of the scoring functions for these argument types is shown in Figure 4.2. One recurrent neural network (RNN) is used to give scores to each hypothesis and lemma by

processing the type of the term, and outputting a final score. A different RNN is then used to process the goal, assigning a score to each token in processes.

As before, blue boxes are inputs; purple boxes are encoded values; green diamonds are outputs, in this case scores for each individual possible argument; and gray circles are computational nodes. The GRU nodes are Gated Recurrent Units [CvMG+14]. The NN node is a feed-forward neural network.

For illustration purposes, Figure 4.2 uses an example to provide sample values. Each token in the goal is an input – in Figure 4.2 the goal is "`not (eq x y)`". The tactic predicted by P_{tac} is also an input – in Figure 4.2 this tactic is "`unfold`". The hypothesis that is heuristically closest to the goal (according to our heuristic from Section 4.1.1) is also an input, one token at a time being fed to a GRU. In our example, let's assume this closest hypothesis is "`y > (x+1)`". The similarity score of this most relevant hypothesis is an additional input – in Figure 4.2 this score is 5.2.

There is an additional RNN (the middle row of GRUs in Figure 4.2) which encodes the goal as a vector of reals. The initial state of this RNN is set to some arbitrary constant, in this case 0.

The initial state of the hypothesis RNN (the third row of GRUs in Figure 4.2) is computed using a feed-forward Neural Network (NN). This feed-forward Neural Network takes as input the tactic predicted by P_{tac}, the goal encoded as a vector of reals, and the similarity score of the hypothesis.

The architecture in Figure 4.2 produces one output score for each token in the goal and one output score for the hypothesis. The highest scoring element will be chosen as the argument to the tactic. In Figure 4.2, the highest scoring element is the "`not`" token, resulting in the proof command "`unfold not`". If the hypothesis score (in our example this score is 8) would have been the highest score, then the chosen argument would be the identifier of that hypothesis in the Coq context. For example, if the identifier was `IHn` (as is sometimes the case for inductive

hypotheses), then the resulting proof command would be "`unfold IHn`".

4.1.3 Combining Tactic and Argument Scores ($\otimes$)

The $\otimes$ operator attempts to provide a balanced combination of tactic and argument prediction, taking both into account even across different tactics. The operator works as follows. We pick the n highest-scoring tactics and for each tactic the m highest-scoring arguments. We then score each proof command by multiplying the tactic score and the argument score, without any normalization. Formally, we can implement this approach by defining $\otimes$ to be multiplication, and by not normalizing the probabilities produced by P_{arg} until all possibilities are considered together.

Because we don't normalize the probabilities of tactics, the potential arguments for a tactic are used in determining the eligibility of the tactic itself (as long as that tactic is in the top n). This forms one of the most important contributions of our work: the argument selection is primary, with the tactic prediction mostly serving to help prune its search space.

4.1.4 Putting It All Together

The overall architecture that we have described is shown in Figure 4.3. The P_{tac} predictor (whose detailed structure is shown in Figure 4.1) computes a distribution over tactic using three features as input: the previous tactic, head constructor of goal, and head constructor of the hypothesis deemed most relevant. Then, for each of the top tactic predicted by P_{tac}, the P_{arg} predictor (whose detailed structure is shown in Figure 4.2) is invoked. In addition to the tactic name, the P_{arg} predictor takes several additional inputs: the goal, the hypotheses in context, and the similarity between each of those hypotheses and the goal. The P_{arg} predictor produces scores for each possible argument (in our case one score for each token in the goal, and one score the single hypothesis). These scores are combined with $\otimes$ to produce an overall scoring of proof

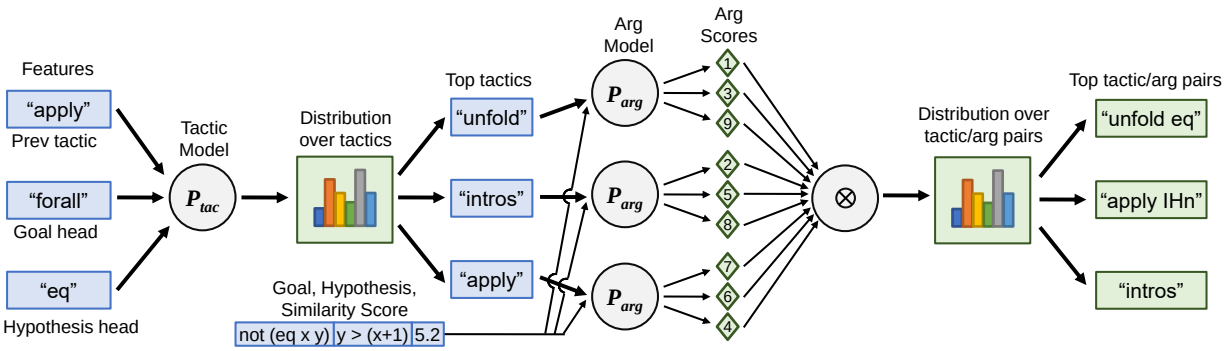


Figure 4.3: The overall prediction model, combining the tactic prediction and argument prediction models.

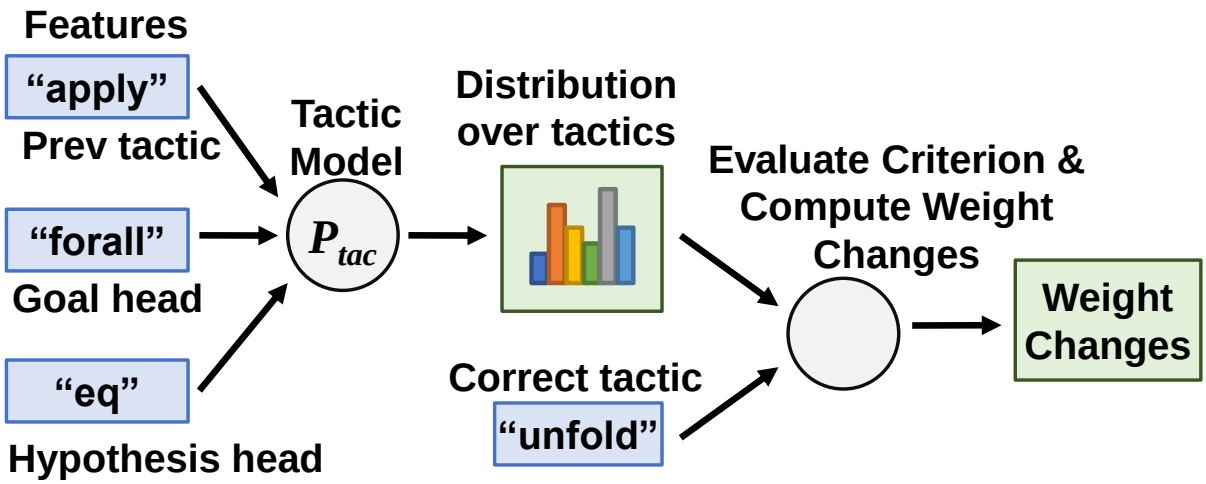


Figure 4.4: The architecture for training the tactic models.

commands.

4.2 Training

4.2.1 Training Architecture

Figure 4.4 shows the training architecture for the tactic predictor, P_{tac} (recall that the detailed architecture of P_{tac} is shown in Figure 4.1). The goal of training is to find weights for the neural network that is found inside the gray P_{tac} circle. Proverbot9001 processes all

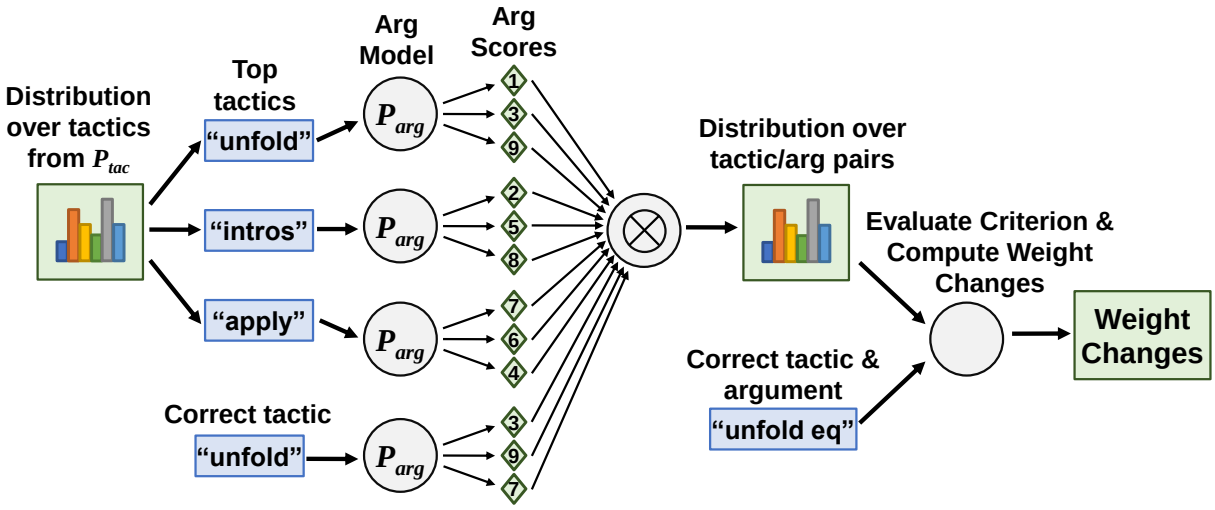


Figure 4.5: The architecture for training the argument models. Note that we inject predicted tactics into the input of the argument model, instead of just using the correct tactic, so that argument scores will be comparable.

the Coq theorems in the training set, and steps through the proof of each of these theorems. Figure 4.4 shows what happens at each step in the proof. In particular, at each step in the proof, Proverbot9001 computes the three features we are training with, and passes these features to the current tactic model to get a distribution over tactics. This distribution over tactics, along with the correct tactic name (from the actual proof), are passed to a module that computes changes to the weights based on the NLLLoss criterion. These changes are batched together over several steps of the proof, and then applied to update the tactic model. Running over all the training data to update the weights is called an epoch, and we run our training over 20 epochs.

Figure 4.5 shows the training architecture for the argument predictor, P_{arg} (recall that the detailed architecture of P_{arg} is shown in Figure 4.2). The goal of training is to find weights for the GRU components in P_{arg}. Here again, Proverbot9001 processes all the Coq theorems in the training set, and steps through the proof of each of these theorems. Figure 4.5 shows what happens at each step in the proof. In particular, at each step in the proof, the current P_{tac} predictor is run to produce the top predictions for tactic. These predicted tactic, along with the correct tactic, are passed to the argument model P_{arg}. To make Figure 4.5 more readable, we do not

show the additional parameters to P_{arg} that where displayed in Figure 4.3, but these parameters are in fact also passed to P_{arg} during training. Note that it is very important for us to inject the tactics predicted by P_{tac} into the input of the argument model P_{arg}, instead of using just the correct tactic name. This allows the scores produced by the argument model to be comparable *across* different predicted tactics. Once the argument model P_{arg} computes a score for each possible argument, we combine these predictions using $\otimes$ to get a distribution of scores over tactic/argument pairs. Finally, this distribution, along with the correct tactic/argument pair is passed to a module that computes changes to the weights based on the NLLLoss criterion. In our main CompCert benchmark the 153402 tactic samples from the training set are processed for 20 epochs.

4.2.2 Learning From Higher-Order Proof Commands

Proof assistants generally have higher-order proof commands, which are tactics that take other proof commands as arguments; in Coq, these are called *tacticals*. One of the most common examples is the (`;`) infix operator which runs the proof command on the right on every sub-goal produced by the tactic on the left. Another example is the `repeat` tactical, which repeats a provided tactic until it fails.

While higher-order proof commands are extremely important for human proof engineers, they are harder to predict automatically because of their generality. While some previous work [YD19] attempts to learn directly on data which uses these higher-order proof commands, we instead take the approach of desugaring higher-order proof commands into first-order ones as much as possible; this makes the data more learnable, without restricting the set of expressible proofs.

For example, instead of trying to learn and predict (`;`) directly, Proverbot9001 has a system which attempts to desugar (`;`) into linear sequences of proof commands. This is not always possible (without using explicit subgoal switching commands), due to propagation of existential

variables across proof branches. Proverbot9001 desugars the cases that can be sequenced, and the remaining commands containing (;) are filtered out of the training set.

In addition to the (;) tactical, there are other tacticals in common use in Coq. Some can be desugared into simpler forms. For example:

- "now <tac>" becomes "<tac>;easy".
- "rewrite <term> by <tac>" becomes "rewrite <term> ; [| <tac>]"
- "assert <term> by <tac>" becomes "assert <term> ; [| <tac>]"

In other cases, like try <tac> or solve <tac>, the tactical changes the behavior of the proof command in a way that cannot be desugared; for these we simply treat the prefixed tactic as a separate, learned tactic. For example, we would treat try eauto as a new tactic.

Chapter 4 is a rewritten version of material published at Machine Learning for Programming Languages (MAPL) 2020. Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner 2020. The dissertation author was the primary author of this paper.

Chapter 5

Reinforcement Learning

5.1 Overview

In problems of supervised learning, the goal is to learn a function from labeled examples of input-output pairs. Models for supervised learning parameterize a function from inputs to outputs and have a procedure to update the parameters from a data set of labeled examples. In contrast to the exampl-based-approximation model of supervised learning, reinforcement learning is a form of semi-supervised learning that acts within an environment to learn a policy of action. As discussed in Section 2.4, a reinforcement model has two parts: a set of states, and a set of actions.

Much like Atari games or the game of Go, proofs in Coq can be modeled as a set of states and actions. In Coq, the set of states is the set of proof states (denoted in the previous section as $\mathcal{S}$), and the actions that move between states are tactics ($\mathcal{T}$). A policy in this modeling takes a context and produces a tactic to run; a successful policy is one which eventually reaches a "Qed" state.

There are three main challenges in applying standard reinforcement learning techniques to this space:

1. **States are complex and difficult to interpret** While in some reinforcement learning problems, states are simply positions of pieces on a board or sprites on a grid, contexts in Coq are structured combinations of goals, hypotheses, and global context. Each of these involves terms whose structures and relationships to one another are non-trivial.

2. **Unbounded and variable actions space** In most reinforcement learning problems, there are a relatively small number of actions that can be taken at any given state; in the game of Go, there are 361 discrete actions that can be taken at the beginning of the game. By contrast, in Coq there are an unbounded number of tactic/argument combinations. In our supervised model, we determined a bounded argument model to reduce the number of possible actions; however this still leaves up to 8,650 possible actions at a given state. To make matters worse, while in most games the set of moves is highly overlapping between states, in Coq the set of moves is entirely dependent on the current context (and thus the hypotheses and goal tokens available as arguments).

3. **Validity of actions is non-trivial** In most games where reinforcement learning is applied, actions are valid or not according to a simple set of rules, even when determining their merit is more complicated. However, the validity of many tactics in Coq depends on complex unification constraints, and therefore the predictor cannot always ensure that it's suggested action is a valid move.

5.2 Model

Our implementation of reinforcement learning in Proverbot9001 uses a standard Q-learning approach at its core, with several additional techniques to tackle the unique challenges of the proof domain. To do so, we learn a function, `q_score`, which takes a state and an action, and attempts to approximate the **Q value** of that action. The Q value of an action is a measure of

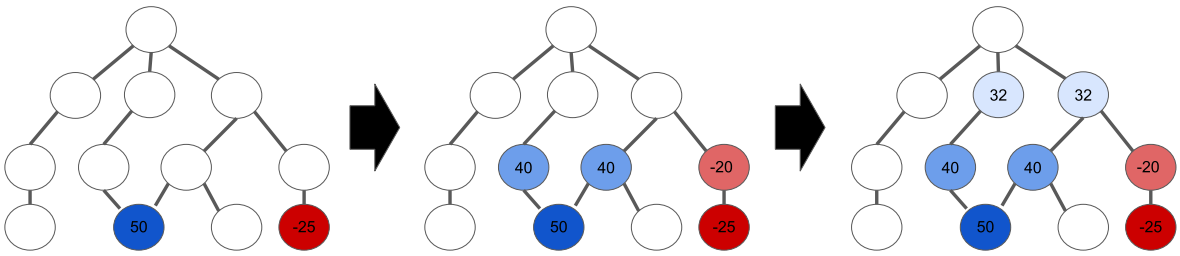


Figure 5.1: Q learning back propagation, grapically. Initially, the Q score just tracks the reward, but in every step, we add the maximum reward of steps we can take, with a time discount.

```
def q_update(memory, q_estimator, predictor, time_discount):
    samples = memory.sample()
    new_qs = []
    for sample in samples:
        predictions = predictor.predict(sample.after_state)
        best_q = max([q_estimator.predict(sample.after_state, action)
                      for action in predictions])
        new_qs.append(best_q * time_discount)
    q_estimator.train(samples, new_qs)
```

Figure 5.2: Q learning psuedocode. At each update step, we sample our memory, and compute an updated q score for each sample by taking the maximum q score of the possible next steps.

the total reward that would result in taking that action, and then continually taking actions which maximize the Q value, where rewards that occur later are discounted by a time factor.

To learn this function, we start with some simple approximation (see Section 5.3.2), and then iteratively improve our estimate with one-step lookahead (see Figure 5.1 and Figure 5.2). That is, we take an action a from state s_0, and compute the resulting state s_1. Then, we evaluate all actions in s_1 using our current Q estimator, discount the result, and add the direct reward of a, to find our new target Q score for action a in s_0.

This approach allows us to update the Q score of individual action/state pairs, but we must also interact with the environment to obtain such pairs. We divide our interactions with the

```
def reinforce(coq, q_estimator, predictor,
              num_episodes, episode_length, time_discount):
    memory = []
    for i in range(num_episodes):
        coq.reset_lemma()
        for t in range(episode_length):
            action = sample_action(coq, predictor, q_estimator)
            coq.run_stmt(action)
            memory.append((coq.state, action))
        q_update(memory, q_estimator, predictor, time_discount)
```

Figure 5.3: A basic reinforcement loop for q learning. This code does not take into account the fallback actions, or actor-learner architecture that we will develop in the following sections.

environment into discrete episodes, where each episode starts at the beginning of a proof, and takes a fixed number of actions, add its experience to an experience buffer at each step. Then, itermittently we sample from the experience buffer, and use that data to update the Q scoring function (see Figure 5.3). Since the actions taken depend on the current Q scoring function, as our function improves we will explore more frequently along the high-reward paths.

5.2.1 Stacked Models

Unfortunately, the space of actions in any possible proof state is large, even when we apply our argument restriction developed in Section 4.1.1. Since our Q scoring function runs once for each state-action pair, it would be computationally infeasible to use it to evaluate every possible action. Instead, we use a stacked model approach, where our supervised tactic prediction model becomes the basis for our reinforced model. In this approach, we first run our supervised model on the context to predict the top 16 tactics, and then use those tactics as our action space for the reinforced predictor.

Fallback actions Unlike in other reinforcement learning contexts, in Coq it is not clear which tactics are even valid actions. If it were, our predictor could produce a single "best" action to

```
...
        for t in range(episode_length):
            actions = sample_actions(coq, predictor, q_estimator)
            successful_action = None
            for action in actions:
                try:
                    old_state = coq.state
                    coq.run_stmt(action)
                    if old_state == coq.state:
                        coq.cancel_last()
                        continue
                    successful_action = action
                    break
                except CoqException:
                    pass
...
```

Figure 5.4: Using a list of actions to account for failed actions. Actions are considered failed if they throw an exception, or don't change the state.

try at every point during reinforcement, and continually following these actions would comprise an episode. Instead, the "best" action produced by the predictor might fail completely, necessitating a fallback to the "second best" action, or third. Therefore, our predictor is designed to not just pick a top action, but instead order all 16 actions, to be tried sequentially (see Figure 5.4).

Sampling actions During reinforcement, we make use of both the stacked model, and the underlying supervised model, to determine what action to take at each step (see Figure 5.5). To sample actions, we first flip a weighted coin to determine whether to exploit (follow our current Q scorer) or explore (attempt to find new paths). If exploit, we order the actions based on our models current q score. If explore, we do a weighted sampling based on the certainties of the underlying supervised predictor model. This ensures that even our exploration doesn't need to consider a large number of actions, and that actions favored by the underlying supervised model are more likely to be explored.

```
def sample_actions(coq, predictor, q_estimator):
    underlying_predictions = predictor.predict(coq.state)
    if random() < exploration_factor:
        actions = order_randomly_by_score(underlying_predictions)
    else:
        actions = sort(underlying_predictions,
                       key=lambda p: q_estimator(coq.state, p))

def order_randomly_by_score(predictions):
    results = []
    for i in len(predictions):
        sample_index = random_index_sample(softmax([p.certainty
                                                    for p in predictions]))
        next_action = predictions.remove(sample_index)
        results.append(next_action)
    return results
```

Figure 5.5: Sampling of actions. We randomly determine whether we are in exploit or explore mode. In exploit mode we order actions by their estimated q scores. In explore mode, we order them randomly, with each action weighted by it's certainty in the underlying predictor.

5.2.2 Encoding actions

Unlike the supervised predictor model developed in Chapter 4, the Q estimator takes as input an action as well as a state. Because of this, we are faced with the new problem of *encoding* actions, rather than just decoding them. We developed two methods of encoding actions, reflected in our two Q estimator models: a naive index-based action encoder, and an RNN action encoder based on our supervised model.

The naive index-based encoder encodes each action as two integers: a unique tactic index, and an argument index. These indices are then embedded into vectors using a word embedding as the first step of the model, so their similarities are learned. While the tactic index is straightforward, and always means the same thing, the argument index is trickier. There is an index for no argument, a range of indices into the goal tokens, and a range of indices into the hypotheses. This model however does not assign meaning to these argument indices, because they are mapping into different goals and sets of hypotheses.

The RNN action encoder encodes the tactic stem the same way, but gives more context to the argument encodings. Instead of merely taking an argument index, it encodes the argument in two parts: an index for the **type** of argument (no argument, goal token, or hypothesis), and then an encoded vector for the term/token referred to by the argument. We encode the arguments using the hypothesis- and goal-encoders that are learned by the supervised model, to make full use of the initial training phase. If the argument type is "no argument", the encoded vector is a vector of zeros.

5.3 Training

5.3.1 Reward structure

Coq proofs are an example of what in reinforcement learning is called "sparse rewards". Instead of receiving intermittent rewards as actors explore the space, the only true reward is found when a proof is finished. This can make it extremely difficult to learn the space, as feedback requires exploring for many steps.

I considered several intermediary reward structures in our development of Proverbot9001, to make the space easier to explore. First, I tried assigning a "complexity" to each proof state, based on the number of characters in it's goal. When a tactic reduced the size of the goal, and thus its complexity, it was given a positive reward. Second, I tried assigning rewards according to the subgoal structure of the proof. When new subgoals were created, the actor would receive negative reward, and when subgoals were solved, the actor would receive positive rewards. Unfortunately, both of these reward structures ended up only obscuring the path to the goal, resulting in fewer proofs completed.

In the final implementation of reinforcement learning in Proverbot9001, I used a simple reward structure. The only positive rewards come from reaching a "Qed", which yields a reward of 50. When all predicted actions are invalid in a particular state, we've reached a "dead-end",

and we assign that state a single action to “Abort.”, which has a reward of -25.

Episode weighting Unfortunately, paths which have positive rewards are still rare compared to paths with no rewards or negative reward. This means that during training, these paths are unlikely to be sampled and learned from. To correct for this bias, we keep track of which rewards were part of an episode which reached “Qed”. At the end of the episode, we add duplicates of these samples into the experience buffer, so that they will be sampled with greater proportion.

5.3.2 Pre-training

The Q learning algorithm requires an initial Q estimator which to iteratively improve. We could initialize the Q estimator randomly to initialize, as many applications of Q learning do. However, since we are already integrating our supervised model into our architecture, we can speed up training by pre-training our Q estimator to an approximation of the supervised model, without interaction with the environment.

To do so, we load an initial set of state-action pairs from our labeled data, and label them with a Q score of the certainty that the supervised predictor would give them, multipllied by the maximum possible reward. So, if our maximum possible reward were 100, and the supervised model ascribed a certainty of .3 to a particular action in a state, we label that action with a Q score of 30. We can quickly build a set of labeled pairs from this process, and use it to “train” the Q estimator to mimic the supervised predictor, as a baseline for the reinforcement process.

5.3.3 Actor-Learner architecture

An additional challenge of applying reinforcement learning to coq proofs is that some actions can take a significant amount of time to execute. While some tactics, like “intros” or “induction” take simple steps to manipulate the context, others like “eapply” rely on potentially complex unification constraints, and some like “firstorder” do search which can take on the order

of minutes in some contexts. Because of this, the time spent spent exploring the environment dominates the training time, significantly slowing down overall progress.

We overcome this barrier by using multiple cores to explore the environment to augment the parallelism inherent in training. These processes interact using an Actor-Learner architecture [MBM+16]. Essentially, instead of holding an experience buffer when exploring environments, actor-threads put their experience samples in a shared queue. A single learner thread then consumes the experience queue, managing the experience buffer and training the Q scoring model (see Figure 5.6). Finally, after each training iteration, the learner thread sends updated weights to the actor threads, to integrate into their future exploration.

5.3.4 Learning from demonstrations

With the sparse rewards in our proof state, for many proofs reinforcement learning will never find a "Qed", and would thus be completely unable to learn from that proof. However, we have multiple sources from which a path to a "Qed" could be available: human written proofs (for proofs where the solutions are available), and proofs found through search (using the supervised predictor). Since our search procedure necessarily searches more exhaustively than our reinforcement episodes, we can even use a partially-learned q estimator to guide search for more solutions, which can then be used to train the q estimator more fully.

Once we have a proof solution, we can use it as a demonstration to more fully learn the Q scores of the proof space. Previous work [HVP+17] found that demonstrations can be used most effectively by traversing backwards along the solution while learning. First, by placing the agent in the second-to-last state of the solution, it will quickly find the action which results a high reward. Then the agent can be interatively moved backwards, at each point only requiring one step to find the state which was given a higher Q-score in the previous step (see Figure 5.7). With this technique, we can focus the reinforcement training along the path of the known solution, and more efficiently explore the Q-score space.

```
def reinforce_learner(samples_queue, predictor, shared_model_namespace):
    experience_buffer = []
    num_samples_retreived = 0
    last_sample_trained = 0
    while True:
        # Allow training anytime between train_every_min
        # and train_every_max, depending on how fast the
        # samples are coming in.
        if num_samples_retreived - last_sample_trained \
                < params.train_every_min:
            experience_buffer.append(samples_queue.get_blocking())
            num_samples_retrieved += 1
            continue
        else:
            try:
                experience_buffer.append(samples_queue.get_timeout(0.1))
                num_samples_retrieved += 1
            except EmptyException:
                continue
            if num_samples_retrieved - last_sample_Trained \
                    < params.train_every_max:
                continue
        # Bound the buffer size through random dropout
        if len(experience_buffer) > args.buffer_max_size:
             experience_buffer = random_sample(memory,
                                               args.buffer_max_size -
                                               args.train_every_max)
        # Train
        last_sample_trained = num_samples_retrieved
        q_estimator = shared_model_namespace.q_estimator
        q_update(experience_buffer, predictor, q_estimator,
                 params.time_discount)
        shared_model_namespace.q_estimator = q_estimator
```

Figure 5.6: The "learner" in the actor-learner architecture. Instead of actors managing their own experience, they pass it to a shared queue, consumed by this learner thread, which runs parallel to all the actors.

```
...
    for i in range(num_episodes):
        coq.reset_lemma()
        for t in range(episode_length):
            if t < len(demonstration) - ((i // params.episodes_per_step)+1):
                actions = [(demonstration[t],
                            certainty_of(predictor, coq.state
                                         demonstration[t]))]
            else:
                actions = sample_actions(coq, predictor, q_estimator)
...
```

Figure 5.7: Learning from demonstrations. This code replaces the normal action sampling when demonstrations are available. At the beginning, the entire demonstration minus the last step is run before any sampling. Every `episodes_per_step` episodes, we remove another step from the end of the demonstration, until every step is sampled (or we run out of episodes).

Chapter 5 is based on ongoing work performed by Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, Sicun Gao, and Sorin Lerner. The dissertation author is the primary author of this work.

Chapter 6

Guided Proof Search

Now that we have explained how we predict a single step in the proof, we describe how Proverbot9001 uses these predictions in a proof search.

In general, proof search works by transitioning the proof assistant into different states by applying proof commands, and backtracking when a given part of the search space has either been exhausted, or deemed unviable. Exhaustive proof search in proof assistants is untenable because the number of possible proof commands to apply is large. Instead, we use the predictor described above to guide the search.

To effectively search the proof space, it is important that we define what parts of the space we're **not** going to search. For every limitation we can make on the search, we can search more advanced proofs in a different direction.

6.1 Bounding the Search

Branch bound The first question when setting up any search tree is, at what factor should that tree branch? In many contexts the answer is a single natural number; however, in the context of Coq proofs, the notion is a little subtler. That's because some predicted tactics can fail

immediately, preventing a branch from being explored any further.

If we apply a strict bound to the number of predictions used, the actual number of explorable branches could be significantly less than that. Furthermore, such a bound would disadvantage predictors which predict difficult-to-apply but useful tactics early on, such as `apply` of a hypothesis. This is despite such predictions rarely hurting search performance when they fail, as they resolve quickly and don't require any further search.

Instead of a strict bound on the number of predictions applied, we have two bounds: a soft bound (default: 5) and a hard bound (default: 10). At each step, our search agent asks the predictor for a number of predictions equal to the hard bound. It then tries these predictions, in order of certainty, until a number of them succeed equal to the soft bound. In this context, we consider success to be both the absense of an error, **and** a change in the context. Some tactics in Coq, like `simpl.`, might not throw an error, but also not change the context.

As an example of this hard bound vs soft bound technique, consider the following example. At a point in the search, the search agent makes predictions with soft-bound two and hard-bound five. The search agent first asks the predictor for the top five predictions, and receives (in order), `eauto`, `simpl`, `apply H`, `apply G`, `exfalso`. It then runs the first two predictions, `eauto` and `simpl`, and finds that the first succeeds, but the second fails to change the context (without an error). Because only one tactic has succeeded thus far, and the soft-bound is two, it continues (after recursively searching the sub-branch of `eauto`) by trying the third prediction, `apply H`. However, `apply H` produces an error message that `H` is not applicable, so it continues and tries `apply G`. `apply G` succeeds, making two successful predictions (the same as the soft-limit), so the search ends, without trying the fifth prediction.

Depth bound In addition to bounding the number of predictions at each step, we also had to bound the depth of our search tree. Unlike the search trees of games, a branch of search in a Coq proof can be arbitrarily long without repetition. However, a naïve depth bound will strongly

penalize proofs that make use of case splitting, excluding whole classes of solutions. To address this, we had to adapt our notion of search "depth" to the structure of Coq proofs (in which a tactic can produce multiple sub-obligations).

Consider for example the following two proofs:

1. `intros. simpl. eauto.`
2. `induction n. eauto. simpl.`

At first glance, it seems that both of these proofs have a depth of three. This means that a straightforward tree search (which is blind to the structure of subproofs) would not find either of these proofs if the depth limit were set to two.

However, there is a subtlety in the second proof above which is important (and yet not visible syntactically). Indeed, the `induction n` proof command actually produces two obligations ("sub-goals" in the Coq terminology). These correspond to the base case and the inductive case for the induction on `n`. Then `eauto` discharges the first obligation (the base case), and `simpl` discharges the second obligation (the inductive case). So in reality, the second proof above really only has a depth of two, not three.

Taking this sub-proof structure into account is important because it allows Proverbot9001 to discover more proofs for a fixed depth. In the example above, if the depth were set to two, and we used a naïve search, we would not find either of the proofs. However, at the same depth of two, a search which takes the sub-proof structure into account would be able to find the second proof (since this second proof would essentially be considered to have a depth of two, not three).

Path-time bound Subgoal-based depth limits are essential for bounding the depth of the search tree, but do not take into account that some actions have significantly greater cost than others. While a tactic like "eauto" is quick to run, one like "firstorder" may take on the order of minutes to complete; in that time, a much larger space of cheaper tactics can be searched. Many existing systems use an overall timeout to bound the search time, where the search will

run normally until a timed interrupt, when it will quit. However, this disproportionally penalizes paths searched later over ones searched earlier.

In Proverbot9001, we developed a *path-time* bounding technique to overcome these limitations. In addition to tracking the number of tactics from the start to reach any node (the path depth), we also track the total amount of time taken by tactics to reach this node (the path time). Each tactic is run using a timeout, which is the minimum of a per-tactic time bound, and the path-time bound minus the amount of time already taken on the current path. By increasing the depth bound, but keeping the path-time bound steady, we can allow Proverbot9001 to search deeply with cheap tactics, without getting slowed down by slow tactics.

6.2 Pruning the Search Tree

Even with these bounds on the search space, exhaustively searching the proof tree is expensive. Fortunately, we don't have to exhaustively search the tree; there is structure to the proof tree that allows us to ignore (or "prune") some branches of the search. We make use of two techniques to prune the search tree, loop pruning, and subgoal pruning.

Loop pruning means we stop the search when we find a proof goal that is at least as hard (by a syntactic definition) as a goal earlier in the history. While in general it is hard to formally define what makes one proof state harder than another, there are some obvious cases which we can detect. A proof state with a superset of the original obligations will be harder to prove, and a proof state with the same goal, but fewer assumptions, will be harder to prove.

To formalize this intuition, we define a relation $\geq$ between states such that $\sigma_1 \geq \sigma_2$ is meant to capture "Proof state σ_1 is at least as hard as proof state σ_2". We say that $\sigma_1 \geq \sigma_2$ if and only if for all obligations O_2 in σ_2 there exists an obligation O_1 in σ_1 such that $O_1 \geq_o O_2$. For obligations O_1 and O_2, we say that $O_1 \geq_o O_2$ if and only if each hypothesis in O_1 is also a hypothesis in O_2, and the goals of O_1 and O_2 are the same.

Since $\geq$ is reflexive, this notion allows us to generalize all the cases above to a single pruning criteria: "proof command prediction produces a proof state which is $\geq$ than a proof state in the history".

Subgoal pruning means that, when backtracking, we do not attempt to find a different proof for an already proven sub-obligation. While in general this can lead to missed proofs because of existential variables (typed holes filled based on context), this has not been an issue for the kinds of proofs we have worked with so far.

Chapter 6 is a rewritten version of material published at Machine Learning for Programming Languages (MAPL) 2020. Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, and Sorin Lerner 2020. The dissertation author was the primary author of this paper.

Chapter 7

Evaluation

7.1 Supervised Learning

This section shows that Proverbot9001 is able to successfully solve many proofs. We also experimentally show that Proverbot9001 improves significantly on the state-of-the-art presented in previous work.

First, in Section 7.1.2, we compare experimentally to previous work on our native benchmark set, by running both Proverbot9001 and the ASTactic [YD19] project on CompCert, in several configurations outlined in the CoqGym paper. After that, in Section 7.1.3, we compare Proverbot9001 to ASTactic [YD19] and another similar work TacTok, on their native benchmark sets, the CoqGym benchmarks, and show that Proverbot9001 is still able to significantly outperform them. Next, in Section 7.1.4, we experiment with using the weights learned from one project to produce proofs in another. Then, in Section 7.1.5, we show the "hardness" of proofs that Proverbot9001 is generally able to complete, using the length of the original solution as proxy for proof difficulty. Finally, in Section 7.1.6 we describe experiments run to test the effectiveness of various subsystems of Proverbot9001.

Except where otherwise noted, experiments were run on two machines. Machine A is

an Intel i7 machine with 4 cores, a NVIDIA Quadro P4000 8BG 256-bit, and 20 gigabytes of memory. Machine B is Intel Xeon E5-2686 v4 machine with 8 cores, a Nvidia Tesla v100 16GB 4096-bit, and 61 gigabytes of memory. Experiments were run using GNU Parallel [Tan11].

During the development of Proverbot9001, we explored many alternatives, including n-gram/bag-of-words representations of terms, a variety of features, and several core models including k-nearest neighbors, support vector machines, and several neural architectures. While we include here some experiments that explore high-level design decisions (such as training and testing on the same projects vs cross project, working with and without solver-based tooling, modifying the search depth and width, and running with and without pre-processing), we also note that in the development of a large system tackling a hard problem, it becomes intractable to evaluate against every possible permutation of every design decision. In this setting, we are still confident in having demonstrated a system that works for the specific problem of generating correctness proof with performance that outperforms the state-of-the-art techniques by many folds.

7.1.1 Summary of Results

Proverbot9001, run using CoqHammer [CK18] and the default configuration, is able to produce proofs for 36% of the theorem statements in CompCert. This represents a 3.1X improvement over the previous state-of-the-art. Without any external tooling, Proverbot9001 can produce proofs for 19.36%, an almost 4X improvement over previous state-of-the-art prediction-based proofs. Our core prediction model is able to reproduce the tactic name from the solution 32% of the time; and when the tactic name is correct, our model is able to predict the solution argument 89% of the time. We also show that Proverbot9001 can be trained on one project and then effectively predict on another project.

7.1.2 Experimental Comparison to Previous Work - CompCert

We tested Proverbot9001 end-to-end by training on the proofs from 162 files from CompCert, and testing on the proofs from 13 different files. On our default configuration, Proverbot9001 solves 19.36% (97/501) of the proofs in our test set.

In addition to running Proverbot9001 on CompCert, we ran the CoqGym [YD19] tool, which represents the state of the art in this area, on the same dataset in several configurations.

To account for differences in training dataset, we ran CoqGym with their original training schema, and also our training schema, and reported the best of the two numbers. CoqGym is intended to be combined with a solver based proof-procedure, CoqHammer [CK18], which is run after every proof command invocation. While our system was not originally designed this way, we compare both systems using CoqHammer, as well as both systems without. We also compared our system to using CoqHammer on the initial goal directly, which simultaneously invokes Z3 [dMB08], CVC4 [BCD$^+$11], Vampire [KV13], and E Prover [Sch13], in addition to attempting to solve the goal using a crush-like tactic [Chl13].

Figure 7.1 shows the proofs solved by various configurations. The configurations are described in the caption. For all configurations, we ran Proverbot9001 with a search depth of 6 and a search width of 3 (see Section 7.1.6). Note that in Figure 7.1 the bars for H, G, and G_H are prior work. The bars P, G+P and G_H+P_H are the ones made possible by our work.

When CoqHammer is not used, Proverbot9001 can complete 4 times the number of proofs that are completed by CoqGym. In fact, even when CoqGym is augmented with CoqHammer Proverbot9001 by itself (without CoqHammer) still completes 62 more proofs, which is a 81% improvement (and corresponds to about 12% of the test set). When enabling CoqHammer in both CoqGym and Proverbot9001, we see that CoqGym solves 76 proofs whereas Proverbot9001 solves 182 proofs, which is a 2.39X improvement over the state of art.

Finally, CoqGym and Proverbot9001 approaches are complementary; both can complete proofs which the other cannot. Therefore, one can combine both tools to produce more solutions

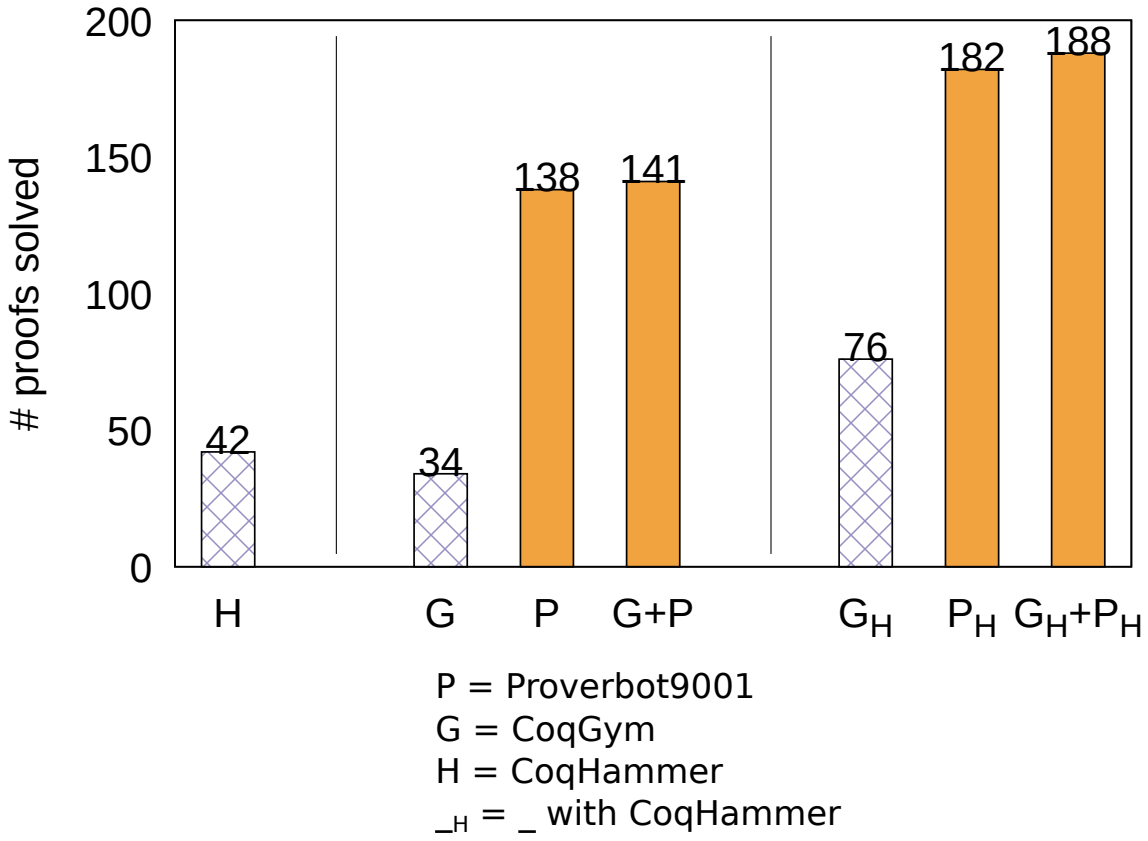


Figure 7.1: A comparison of Proverbot9001 and CoqGym's abilities to complete proofs. H stands for CoqHammer by itself, as a single invocation; G stands for CoqGym by itself; P stands for Proverbot9001 by itself; G+P stands for the union of proofs done by G or P; G_H stands for CoqGym with CoqHammer; P_H stands for Proverbot9001 with CoqHammer; G_H+P_H stands for the union of proofs done by G_H or P_H.

than either alone. Combining CoqGym and Proverbot9001, without CoqHammer, allows us to complete 141/507 proofs, a proof success rate of 27%. Combining Proverbot9001 and CoqGym, each with CoqHammer, allows us to solve 188/507 proofs, a success rate of 37%. It's important to realize that, whereas the prior state of the art was CoqGym with CoqHammer, at 76 proofs, by combining CoqGym and Proverbot9001 (both with CoqHammer), we can reach a grand total of 188 proofs, which is a 2.47X improvement over the prior state of art.

7.1.3 Experimental Comparison to Previous Work - CoqGym

While the above section shows that we outperform the ASTactic model on our CompCert benchmarks, it doesn't determine whether we are overadapted to that particular set. To show that our success is not dependent on a particular set of benchmarks, we also compared the tools on the CoqGym benchmark set. Using this existing benchmark set also allowed us to compare to another previouw work, TacTok [FBG20], Both the ASTactic model and a later work, TacTok, were developed on the CoqGym benchmark set.

The CoqGym benchmark set is split into "testing" and "training" by project. There are 69 training projects and 27 testing projects total. However, due to the time between the projects development, Coq versions had varied enough to cause compatibility issues with several of the projects. With manual effort, we were able to get 58 of the 69 training projects to train under Proverbot9001, and 25 of the 27 testing projects. For the training data, since we were able to train with 85% of the training projects, and trained with strictly less data than CoqGym, we believe our training is comparable. For testing, we were able to test on almost every project; results are largely compared per-project, and we only counted results on the projects that all three tools could run on.

In this comparison, we did not include hammer automation in the results of any of the tools. These experiments were run on a third machine, Machine C. Machine C is a Intel(R) Xeon(R) machine with 80 cores, with a NVIDIA GV102 and 377 GB of memory.

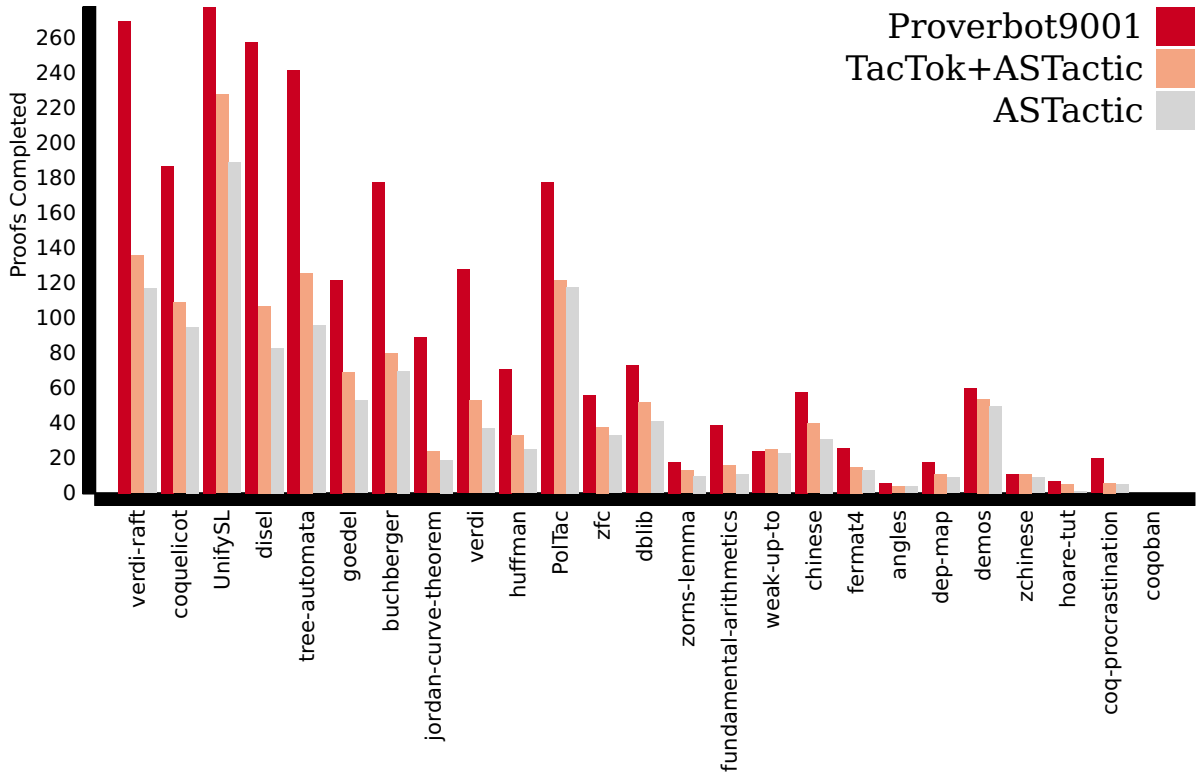


Figure 7.2: The results of running ASTactic, TacTok+ASTactic, and Proverbot9001 on the CoqGym test projects. Projects are ordered largest-to-smallest, left-to-right.

Our results show that, of the 11,729 proofs in the test set, ASTactic alone can complete 1142 (9.7%), TacTok with ASTactic can complete 1377 (11.7%), and Proverbot9001 alone can complete 2417 proofs(20.6%). That is, Proverbot9001 can complete over 2X as many proofs as CoqGym's ASTactic model, on the CoqGym dataset, and 1.75X as many proofs as TacTok combined with the ASTactic model.

In Figure 7.2, we show the per-project results of this comparison. As you can see, the larger projects show the most improvement by Proverbot9001, where as on the smaller projects the results are more similar.

7.1.4 Cross-Project Predictions

To test Proverbot9001's ability to make use of training across projects, we used the weights learned from CompCert, and ran Proverbot9001 in its default configuration on fourteen other Coq projects from the Coq Contrib collection. Seven of the fourteen projects are formalizations of mathematical ideas, and the other seven are libraries for verified programming. For these experiments, we did not make use of solver-based automation.

Math `concat` [1] is a library of constructive category theory proofs, which showcases Coq proofs of mathematical concepts instead of program correctness. The `concat` library is made of 514 proofs across 105 files; Proverbot9001 was able to successfully produce a proof for 91 (17.7%) of the extracted theorem statements.

`zfc` [2] is a formalization of set theory made of 241 proofs across 78 files; 41 (17.01%) were successfully completed.

`angles` [3] is a formalization of the theory of oriented angles of non-zero vectors, made up of 89 proofs across 6 files. Proverbot9001 was able to successfully produce a proof for 6 (6.74%)

[1] https://github.com/coq-contribs/concat
[2] https://github.com/coq-contribs/zfc
[3] https://github.com/coq-contribs/angles/

lemmas.

`chinese` [4] and `zchinese` [5] are two different formalizations of the chinese remainder theorem in Coq. They have 127 and 39 proofs respectively, and Proverbot9001 can re-prove 50 (39.37%) and 9 (23.08%) of them.

`UnifySL` [6] is a Coq library of mathematical concepts for working with finite model methods. It has 1058 proofs, and Proverbot9001 can re-prove 218 (20.60%) of them.

Programming `float` is a formalization of floating point numbers, made of 742 proofs across 38 files; Proverbot9001 was able to successfully produce a proof for 100 (13.48%) proofs.

`buchberger` [7] is a verified implementation of Buchbergers algorithm. It has 673 proofs, across 28 files, of which Proverbot9001 was able to successfully re-prove 130 (19.32%).

`dblib` [8] is a Coq library for working with de Bruijn indices, a fundamental concept in programming languages. It has 189 proofs, of which Proverbot9001 is able to prove 55 (29.10%).

`depmap` [9] is an implementation of dependent maps in Coq. It has 90 proofs, of which Proverbot9001 could solve 19 (21.11%).

`disel` [10] is a separation logic for compositional proofs of distributed systems. It has 818 proofs, and Proverbot9001 can solve 194 of them (23.72%).

`hoare-tut` [11] is a tutorial for Hoare Logic, a logic for proving things about imperative programs. It has 24 proofs, of which Proverbot9001 can find solutions for 7 (29.17%).

`huffman` [12] is a Coq proof of the correctness of Huffman coding, a method of constructing minimum redudancy codes published in 1952. It has 298 proofs, of which Proverbot9001 can

[4] https://github.com/coq-contribs/chinese
[5] https://github.com/coq-contribs/zchinese
[6] https://github.com/QinxiangCao/UnifySL
[7] https://github.com/coq-community/buchberger
[8] https://github.com/coq-community/dblib
[9] https://github.com/coq-contribs/dep-map
[10] https://github.com/DistributedComponents/disel
[11] https://github.com/coq-community/hoare-tut
[12] https://github.com/coq-community/huffman

re-prove 51 (17.11%).

`weak-up-to` [13] is a Coq formalization of up-to techniques for weak bisimulation. It has 150 proofs, of which Proverbot9001 can successfully re-prove 21 (14%).

Summary In total the projects we ran on had 5052 proofs (2068 math, 2984 programming). The projects had a variety of sizes, and a variety of solve rates, from only 6.74% (`angles`) to 39.37% (`chinese`). On average, Proverbot9001 could solve 19.6% of the proofs with little variation between the math and programming projects (20.06% math, 19.33% programming).

The comparable number for CompCert was 19.36%.

These results demonstrate not only that Proverbot9001 can operate on proof projects in a variety of domains, but more importantly that it can effectively transfer training from one project to another. This would allow programmers to use Proverbot9001 even in the initial development of a project, if it had been previously trained on other projects.

7.1.5 Original Proof Length vs Completion Rate

In Figure 7.3 and Figure 7.4, we plot a histogram of the original proof lengths (in proof commands) vs the number of proofs of that length. We break down the proofs by (from bottom to top) number we solve, number we cannot solve but still have unexplored nodes, and number run out of unexplored nodes before finding a solution. Note that for the second class (middle bar), it's possible that increasing the search depth would allow us to complete the proof. Figure 7.3 shows proofs of length 10 or below, and Figure 7.4 shows all proofs, binned in sets of 10.

There are several observations that can be made. *First*, most original proofs in our test set are less than 20 steps long, with a heavy tail of longer proofs. *Second*, we do better on shorter proofs. Indeed, 51% (256/501) of the original proofs in our test set are ten proof commands or shorter, and of those proofs, we can solve 35% (89/256), compared to our overall solve rate of

[13] https://github.com/coq-contribs/weak-up-to

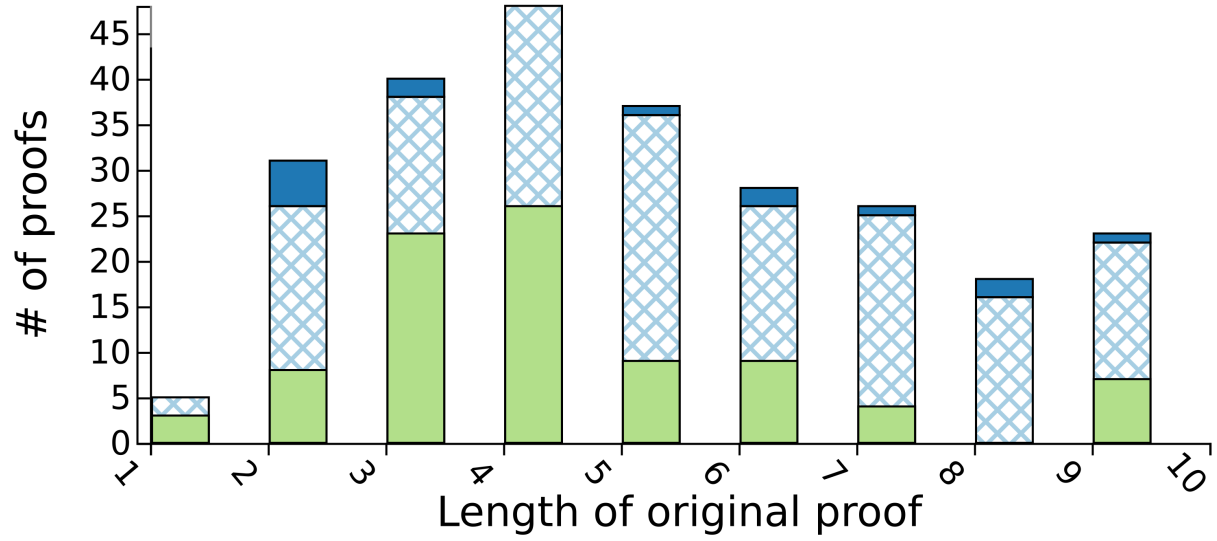


Figure 7.3: A histogram plotting the original proof lengths in proof commands vs number of proofs of that length, in three classes, for proofs with length 10 or less. From bottom to top: proofs solved, proofs unsolved because of depth limit, and proofs where our search space was exhausted without finding a solution.

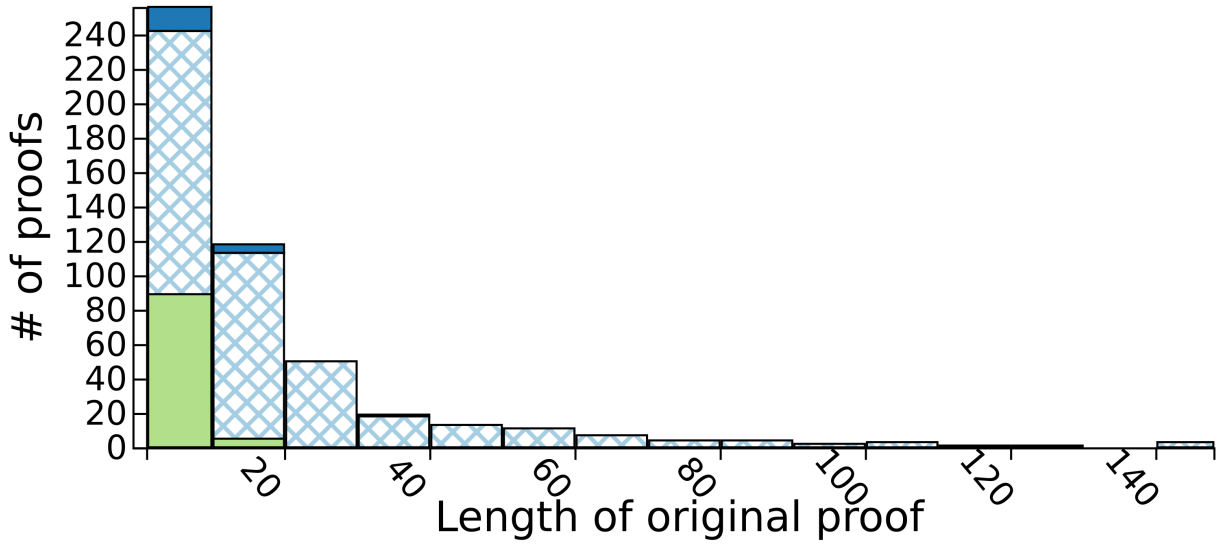


Figure 7.4: A histogram plotting the original proof lengths in proof commands vs number of proofs of that length, in three classes. From bottom to top: proofs solved, proofs unsolved because of depth limit, and proofs where our search space was exhausted without finding a solution. Note that most proofs are between 0 and 10 proof commands long, with a long tail of much longer proofs.

19.36% (97/501). *Third*, we are in some cases able to handle proofs whose original length is longer than 10. Indeed, 7 of the proofs we solve (out of 79 solved) had an original length longer than 10. In fact, the longest proof we solve is originally 25 proof commands long; linearized it's 256 proof commands long. Our solution proof is 267 (linear) proof commands long, comparable to the original proof, with frequent case splits. The depth limit for individual obligations in our search was 6 in all of these runs.

7.1.6 Evaluation of Components

Individual Prediction Accuracy

We want to measure the effectiveness of the predictor subsystem that predicts proof command pairs (the *P* function defined in Section 4.1). To do this, we broke the test dataset down into individual (linearized) proof commands, and ran to just before each proof command to get its prediction context. Then we fed that context into our predictor, and compared the result to the proof command in the original solution. Of all the proof commands in our test dataset, we are able to predict 28.66% (3784/13203) accurately. This includes the correct tactic and the correct argument. If we only test on the proof commands which are in Proverbot9001's prediction domain, we are able to predict 39.25% (3210/8178) accurately.

During search, our proof command predictor returns the top N tactics for various values of N, and all of these proof commands are tried. Therefore, we also measured how often the proof command in the original proof is in the top 3 predictions, and the top 5 predictions. For all proof commands in the data set, the tactic in the original proof is in our top 3 predictions 38.93% of the time, and in our top 5 predictions 42.66% of the time. If we restrict to proof commands in Proverbot9001's prediction domain, those numbers are 52.17% and 60.39%.

Argument Accuracy

Our argument prediction model is crucial to the success of our system, and forms one

of the main contributions of our work. To measure its efficacy at improving search is hard, because it's impossible to separate its success in progressing a proof from the success of the tactic predictor. However, we can measure how it contributes to individual prediction accuracy.

On our test dataset, where we can predict the full proof command in the original proof correctly 28.66% of the time, we predict the tactic correctly but the argument wrong 32.24% of the time. Put another way, when we successfully predict the tactic, we can predict the argument successfully with 89% accuracy. If we only test on proof commands within Proverbot9001's prediction domain, where we correctly predict the entire proof command 39.25% of the time, we predict the name correctly 41.01% of the time; that is, our argument accuracy is 96% when we get the tactic right. It's important to note, however, that many common tactics don't take any arguments, and thus predicting their arguments is trivial.

Completion Rate in Proverbot9001's Prediction Domain

Proverbot9001 has a restricted model of proof commands: it only captures proof commands with a single argument that is a hypothesis identifier or a token in the goal. As result, it makes sense to consider Proverbot9001 within the context of proofs that were originally solved with these types of proof commands. We will call proofs that were originally solved using these types of proof commands *proofs that are in Proverbot9001's prediction domain*. There are 79 such proofs in our test dataset (15.77% of the proofs in the test dataset), and Proverbot9001 was able to solve 48 of them.

What is interesting is that Proverbot9001 is able to solve proofs that are *not* in its prediction domain: these are proofs that were originally performed with proof commands that are *not* in Proverbot9001's domain, but Proverbot9001 found another proof of the theorem that *is* in its domain. This happened for 49 proofs (out of a total of 97 solved proofs). Sometimes this is because Proverbot9001 is able to find a simpler proof command which fills the exact role of a more complex one in the original proof; for instance, `destruct (find_symbol ge id)` in an

original proof is replaced by `destruct find_symbol` in Proverbot9001's solution. Other times it is because Proverbot9001 finds a proof which takes an entirely different path than the original. In fact, 31 of Proverbot9001's 97 found solutions are shorter than the original. It's useful to note that while previous work had a more expressive proof command model, in practice it was unable to solve as many proofs as Proverbot9001 could in our more restricted model.

Together, these numbers indicate that the restricted tactic model used by Proverbot9001 does not inhibit its ability to solve proofs in practice, even when the original proof solution used tactics outside of that model.

Data Transformation

Crucial to Proverbot9001's performance is its ability to learn from data which is not initially in its proof command model, but can be transformed into data which is. This includes desugaring tacticals like `now`, splitting up multi-argument tactics like `unfold a, b` into single argument ones, and rearranging proofs with semicolons into linear series of proof commands. To evaluate how much this data transformation contributes to the overall performance of Proverbot9001, we disabled it, and instead filtered the proof commands in the dataset which did not fit into our proof command model.

With data transformation disabled, and the default search width (5) and depth (6), the proof completion accuracy of Proverbot9001 is 15.57% (78/501 proofs). Recall that with data transformation enabled as usual, this accuracy is 19.36%. This shows that the end-to-end performance of Proverbot9001 benefits greatly from the transformation of input data, although it still outperforms prior work (CoqGym) without it.

When we measure the individual prediction accuracy of our model, trained without data transformation, we see that its performance significantly decreases (16.32% instead of 26.77%), demonstrating that the extra data produced by preprocessing is crucial to training a good tactic predictor.

Search Widths and Depths

Our search procedure has two main parameters, a *search width*, and a *search depth*. The *search width* is how many predictions are explored at each context. The *search depth* is the longest path from the root a single proof obligation state can have.

To explore the space of possible depths and widths, we varied the depth and width, on our default configuration without external tooling. With a search width of 1 (no search, just running the first prediction), and a depth of 6, we can solve 5.59% (28/501) of proofs in our test dataset. With a search width of 2, and a depth of 6, we're able to solve 16.17% (81/501) of proofs, as opposed to a width of 3 and depth of 6, where we can solve 19.36% of proofs.

To explore variations in depth, we set the width at 3, and varied depth. With a depth of 2, we were able to solve 5.19% (26/501) of the proofs in our test set. By increasing the depth to 4, we were able to solve 13.97% (70/501) of the proofs in our test set. At a depth of 6 (our default), that amount goes up to 19.36% (97/501).

7.1.7 Proof Examples

Longest Generated Proof

The longest proof discovered by Proverbot9001 is a solution to the `diff_sym` lemma in CompCert's Locations module. This module says that disjointness of locations, which are either registers or regions of memory, is symmetric; meaning that if location $l1$ doesn't overlap with location $l2$, then location $l2$ doesn't overlap with $l1$. The discovered solution is 106 commands long, involving many case splits (see Figure 7.5c).

The original version of the proof is much shorter at five tactics (see Figure 7.5a). Part of that is that it uses the `;` tactical for repeated tactic invocation; when the proof is unfolded, it is twelve tactics long (see Figure 7.5b). The other difference between the proofs is that where the original case splits on both locations, and unfolds a function symbol, the generated proof

```
Lemma diff_sym:
  forall l1 l2, diff l1 l2 -> diff l2 l1.
Proof.
  destruct l1; destruct l2; unfold diff; auto.
  intuition.
Qed.
```

(a) The original proof of `diff_sym` in CompCert.

```
Lemma diff_sym:
  forall l1 l2, diff l1 l2 -> diff l2 l1.
Proof.
  destruct l1.

  destruct l2.
  unfold diff. auto.
  unfold diff. auto.

  destruct l2.
  unfold diff. auto.
  unfold diff. auto.

  intuition.
Qed.
```

(b) The unfolded original proof of `diff_sym` in CompCert.

```
Lemma diff_sym:
  forall l1 l2, diff l1 l2 -> diff l2 l1.
Proof.
  induction l1. intros. red.
  destruct r.

  destruct l2. eauto. eauto.
  destruct l2. eauto. eauto.
  (* Repeats 30 more times *)

  simpl. intros.
  destruct l2. eauto. simpl. intuition.
Qed.
```

(c) The proof of `diff_sym` generated by Proverbot9001.

Figure 7.5: Three proofs of `diff_sym`

case splits on both locations *and* the register name within the first location. This results in the generated proof being significantly longer.

However, because so much of the length of the proof is due to case splitting, our subgoal-depth bounding mechanism allows us to find this 103 tactic proof without adjusting our normal depth limit.

Short failed proofs Though Proverbot9001 is powerful, there are still simple proofs for which it is unable to find any solution.

In Figure 7.6, we show three simple proofs that Proverbot9001 is not able to solve. All of these proofs come from CompCert's `backend/Selectionproof.v` file, which proves the correctness of instruction selection.

In the first proof, `symbols_preserved` in Figure 7.6a, the proof is a single line. In this proof a literal term is given as the proof, instead of invoking any tactics. Because Proverbot9001s proof synthesis is built around tactic prediction, proofs of this kind cannot be generated.

For the section proof, `functions_translated` in Figure 7.6b, the original proof is six tactics long, and involves applying the `Genv.find_funct_match` lemma. Although it's lemma sources are configurable, by default Proverbot9001 only uses lemmas defined in the current file for consideration as tactic arguments. In this setup, it's not possible for Proverbot9001 to apply this particular lemma, solving this proof.

Finally, `stackspace_functions_translated` in Figure 7.6c neither uses a proof term nor applies any external lemmas. However, it uses the monadInv tactic, a custom tactic in CompCert. While Proverbot9001 can make use of custom tactics that are present in its training data (and often does in other solutions), it is not generally as effective at using more specialized tactics that appear less frequently.

Section 7.1 is a rewritten and updated version of material published at Machine Learning for Programming Languages (MAPL) 2020. Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul,

```
Lemma symbols_preserved:
  forall (s: ident), Genv.find_symbol tge s = Genv.find_symbol ge s.
Proof (Genv.find_symbol_match TRANSF).
```

(a) `symbols_preserved`

```
Lemma functions_translated:
  forall (v v': val) (f: Cminor.fundef),
  Genv.find_funct ge v = Some f ->
  Val.lessdef v v' ->
  exists cu tf, Genv.find_funct tge v' = Some tf /\
                match_fundef cu f tf /\ linkorder cu prog.
Proof.
  intros.
  inv H0.
  eapply Genv.find_funct_match.
  eauto.
  eauto.
  discriminate.
Qed.
```

(b) `functions_translated`

```
Lemma stackspace_function_translated:
  forall dm hf f tf, sel_function dm hf f = OK tf ->
    fn_stackspace tf = Cminor.fn_stackspace f.
Proof.
  intros. monadInv H. auto.
Qed.
```

(c) `stackspace_function_translated`

Figure 7.6: Three simple proofs that Proverbot9001 cannot solve.

and Sorin Lerner 2020. The dissertation author was the primary author of this paper.

7.2 Reinforcement Learning

In this section, we explore the performance of the action evaluator and the resulting reinforced predictor. All experiments in this section were carried out on a Linux machine with 8 Nvidia GeForce RTX 2080 Ti's, and an Intel Xeon Gold 6230 CPU with 40 cores.

7.2.1 Simplfied proof domain

For this evaluation, I wanted to consider only a simple subset of proofs, which could be solved by continued application of a few basic tactics. I start with the argument model from the supervised model: no multiple arguments, only a single identifier, must be a token in the goal or the name of a hypothesis. Then, I measured the tactics which occur in the most proofs.

Figure 7.7 shows the 25 most common tactics in our test set. For each one, the second column shows how many proofs would be eliminated if we disallowed those with the tactic in the original solution. The third column shows the total number of occurances of each tactic. Note that the more common tactics in terms of total number of occurances are not necessarily present in a larger number of proofs. For instance, the `intro` tactic is present in a greater number of proofs than `auto`, but auto is used a greater number of times total.

For various values of N, I restrict the proofs to only those which are made up of tactics from the top N of above. Figure Figure 7.8, shows the results.

The blue bar shows the total number of proofs in that simplicity category. For example, the first visible blue bar, at 4, represents the proofs using only the top 4 most present tactics, `auto` (and `eauto`), `apply` (and `eapply`), `intro` (and `intros`), and `destruct`. There are 8 proofs that fit into this classification, and both supervised learning and reinforcement learning can solve 5 of them. For the reinforcement result, the action evaluator reinforced on all the proofs in this

Tactic	Proofs Including	Occurances
intro(s)	460	957
(e)auto	316	1380
(e)apply	315	1073
destruct	249	702
unfold	219	371
(e)rewrite (\<-)	204	685
simpl	187	348
red	105	162
induction	86	90
split	77	195
inv	76	211
(e)exists	71	112
subst	68	107
assert	66	125
simpl in	63	92
(e)constructor	61	266
TrivialExists	52	101
congruence	52	64
(e)exploit	48	114
generalize	45	60

Figure 7.7: The 25 most common tactics in our Compcert test dataset, with how many proofs include them. The first column shows the tactic name; the second shows the number of proofs whose original solution includes that tactic at least once; and the third column is the total number of occurances of that tactic in the dataset.

class (without any labels). The resulting estimator was combined with the supervised model, to produce a reinforced model, which starts with the supervised models predictions, and re-grades the top k using the action evaluator. The top 5 of this re-grading are used for search.

7.2.2 Number of Episodes vs Solve Rate

Next, I wanted to explore how the number of episodes used for reinforcement affects the solve rate. We would expect that at low numbers of episodes, the pre-training, which trains the action evaluator to reproduce the certainties of the underlying supervised predictor, would dominate. As more episodes are added, the experience begins to dominate the pretraining. I took the class of proofs using the top 6 tactics (22 proofs), and reinforced on them using 8, 16, 32, 64, and 128 episodes. You can see the results in Figure 7.9.

As you can see from the graph, at 0 episodes, the reinforced model mirrors the supervised one, and is thus able to solve 16 proofs (just like the supervised model). As the model trains, at 8 and 16 epochs, it diverges from the supervised model and begins to degrade in performance. At 32 epochs, it begins to learn its own estimation function, and it's performance improves. And by 64 epochs, it is once again on par with the supervised system.

Section 7.2 is based on ongoing work performed by Alex Sanchez-Stern, Yousef Alhessi, Lawrence Saul, Sicun Gao, and Sorin Lerner. The dissertation author is the primary author of this work.

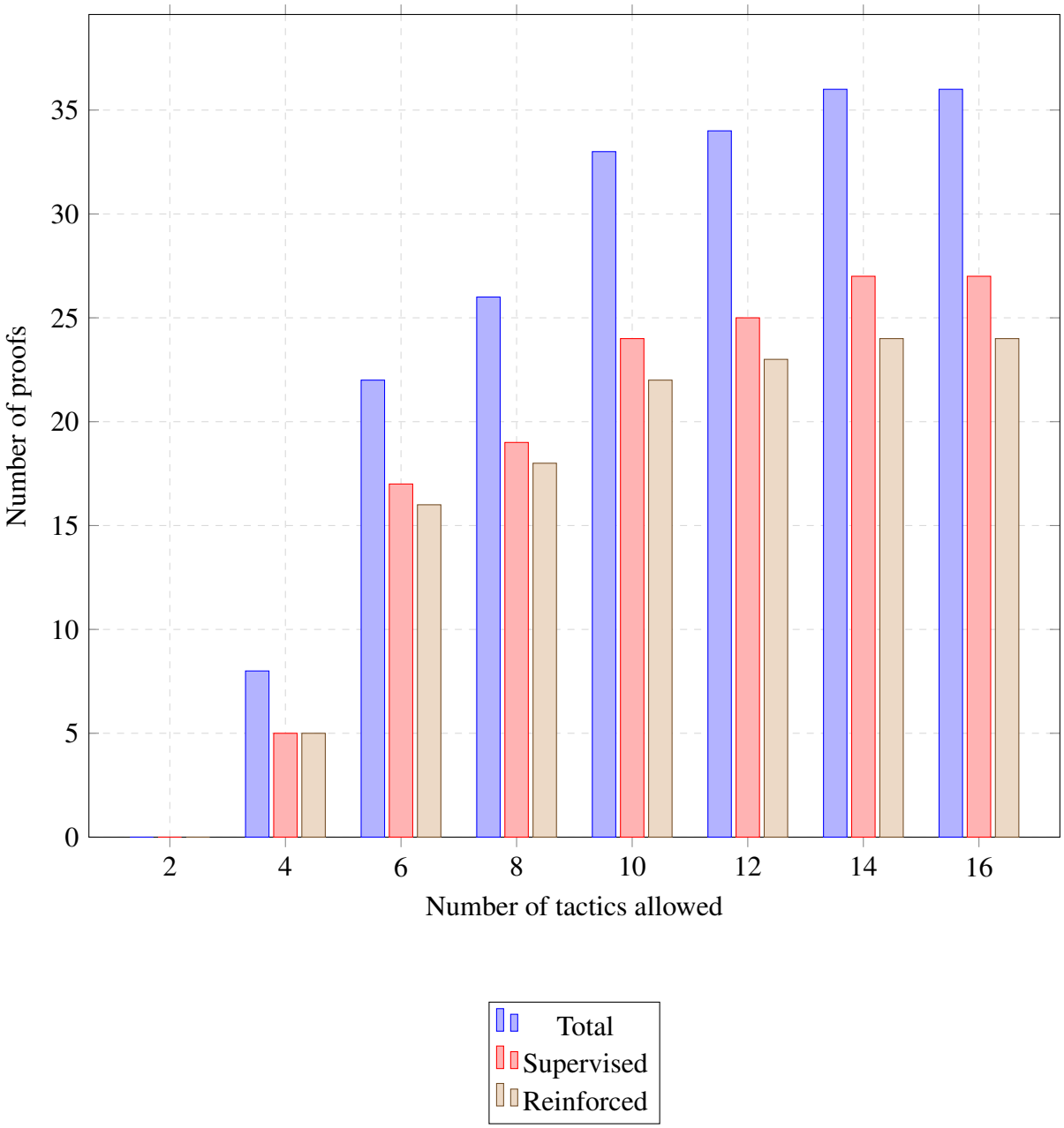


Figure 7.8: The total proofs, proofs solved by supervised learning, and proofs solved by reinforced learning, for simple proofs containing the top N tactics.

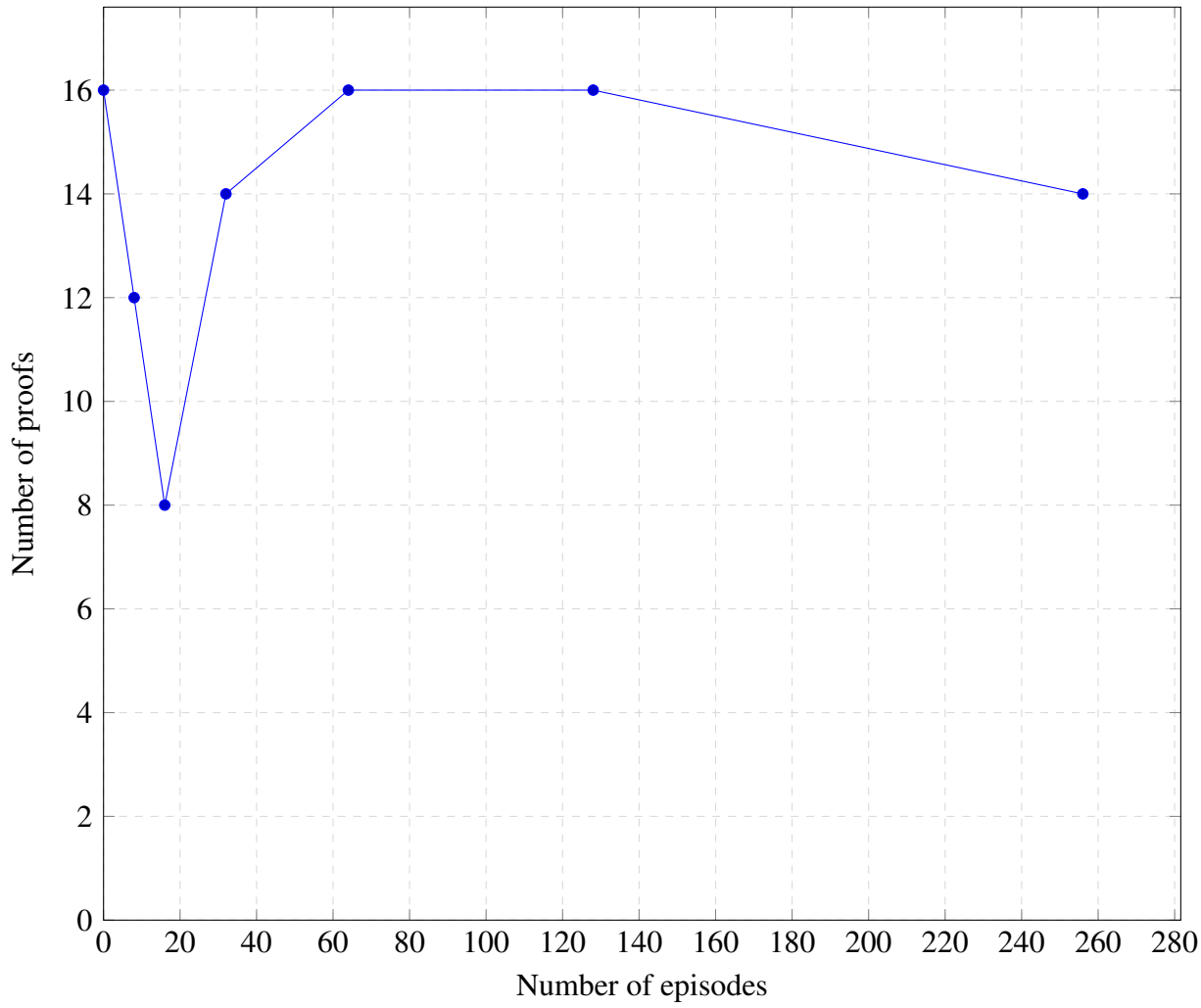


Figure 7.9: The number of episodes used for reinforcement, versus the number of proofs solved. Uses the 22 proofs which use only the top 6 tactics.

Chapter 8

Related Work

8.1 Program Synthesis

Program Synthesis is the automatic generation of programs from a high-level specification [Gul10]. This specification can come in many forms, the most common being a logical formula over inputs and outputs, or a set of input-output examples. Programs generated can be in a variety of paradigms and languages, often domain-specific. Our tool, Proverbot9001, is a program synthesis tool that focuses on synthesis of proof command programs.

Several program synthesis works have used types extensively to guide search. Some work synthesizes programs purely from their types [GKKP13], while other work uses both a type and a set of examples to synthesize programs [OZ15, FOWZ16]. In Proverbot9001, the programs being synthesized use a term type as their specification, however, the proof command program itself isn't typed using that type, rather it must generate a term of that type (through search).

Further work in [LAR17] attempts to learn from a set of patches on GitHub, general rules for inferring patches to software. This work does not use traditional machine learning techniques, but nevertheless learns from data, albeit in a restricted way.

8.2 Supervised Learning for Code

Machine learning for modeling code is a well explored area [ABDS17], as an alternative to more structured methods of modeling code. Several models have been proposed for learning code, such as AST-like trees [MLJ+14], long-term language models [DTP16], and probabilistic grammars [BRV16]. Proverbot9001 does not attempt to be so general, using a model of programs that is specific to its domain, allowing us to capture the unique dependencies of proof command languages. While the model is simple, it is able to model real proofs better than more general models in similar domains (see Section 7.1.2). Machine learning has been used for various tasks such as code and patch generation [ABDS17, BRV16, DTP16], program classification [MLJ+14], and learning loop invariants [GNMR16].

8.3 Supervised Learning for Proofs

While machine learning has previously been explored for various aspects of proof writing, we believe there are still significant opportunities for improving on the state-of-the-art, getting closer and closer to making foundational verification broadly applicable.

More concretely, work on machine learning for proofs includes: using machine learning to speed up automated solvers [BBV18], developing data sets [KCS17, YD19, BLR+19], doing premise selection [LISK17, ACI+16], pattern recognition [KL12], clustering proof data [KHG12], learning from synthetic data [HDSS18], interactively suggesting tactics [KHG12, HK14].

CoqGym's ASTactic model [YD19] attempts to model proofs with a fully general proof command and term model expressing arbitrary AST's. We experimentally compare Proverbot9001's ability to complete proofs to that of ASTactic in detail in Section 7.1.2 There are also several important conceptual differences. *First*, the argument model in ASTactic is not as expressive as the one in Proverbot9001. ASTactic's argument model can predict a hypothesis name, a number between 1 and 4 (which many tactics in Coq interpret as referring to binders,

for example `induction 2` performs induction on the second quantified variable), or a random (not predicted using machine learning) quantified variable in the goal. In contrast, the argument model in Proverbot9001 can predict any token in the goal, which subsumes the numbers and the quantified variables that ASTactic can predict. Most importantly because Proverbot9001's model can predict symbols in the goal, which allows effective unfolding, for example "`unfold eq`". *Second*, in contrast to ASTactic, Proverbot9001 uses several hand-tuned features for predicting proof commands. One key example is the previous tactic, which ASTactic does not even encode as part of the context. *Third*, ASTactic's treatment of higher-order proof commands like "`;`" is not as effective as Proverbot9001's. While neither system can predict "`;`", Proverbot9001 learns from "`;`" by linearizing them, whereas ASTactic does not.

TacTok [FBG20] improves upon ASTactic's model by including sequence modeling of the previous steps in the proof. It contains two seperate models, Tac which includes only common Coq tactic names in it's encoding of the previous steps, and Tok which includes all tokens in the previously run tactics. These models are combined in an ensemble fashion to produce proofs, with both models being run on all problems. Proverbot9001s previous tactic feature contains a subset of the previous tactics information used by TacTok, but could likely be improved by a more detailed encoding like the one used by TacTok. Currently, proof completion results of TacTok lag significantly behind those of Proverbot9001

There is also a recent line of work on doing end-to-end proofs in Isabelle/HOL and HOL4 [GKU17, BLR+19, PLR+19]. This work is hard to experimentally compare to ours, since they use different benchmark sets, proof styles, and proof languages. Their most recent work [PLR+19] uses graph representations of terms, which is a technique that we have not yet used, and could adapt if proven successful.

Finally, there is also another approach to proof generation, which is to generate the term directly using language translation models [SIS17], instead of using tactics; however this technique has only been applied to small proofs due to its direct generation of low-level proof

term syntax.

8.4 Reinforcement Learning for Proofs

Several previous works have attempted to use reinforcement learning to aid in theorem proving, both in the automated and interactive settings.

In automated theorem proving, two previous works have replaced the iterative deepening strategy, where proofs are searched for in a breadth-first manner, with a reinforcement-learned heuristic. Work by Zombori et al [ZCM+19] applied reinforcement learning (and curriculum learning) to LeanCoP/FCoP proofs, but only for simple arithmetic statements. rlCoP [KUMO18] improved on this work on FCoP proofs, expanding it to a broader class of theorems, from the Mizar40 mathematical theorem dataset.

Works applying reinforcement learning to interactive theorem proving are more similar to the work covered in this thesis. TacticToe [GKU17] tried reinforcement learning for HOL4, but reported no significant increase in performance; followup work bringing the same techniques to Coq [BUG20] didn't attempt it again. However, the HOList system [BLR+19] implemented reinforcement learning on Issabelle/HOL proofs with an Actor-Learner architecture, and reported some improvement.

Chapter 9

Conclusion

9.1 Lessons Learned

Over the course of developing Proverbot9001, we learned several key lessons about building ML tools for proof search.

Simplify training data The Coq proof assistant is a complex system, and it has been constantly evolving since it's release over thirty years ago. In such a system, there are multiple ways to do many common tasks. Additionally, Coq includes several features intended to improve ergonomics and help human provers efficiently use Coq. But these features only make things harder for automated methods.

Over the course of development of Proverbot9001, we learned that variety in the training data significantly hurts our ability to train an accurate model. So, to improve our models accuracy, it is necessary to normalize the training data into a simpler form. In Section 4.2.2, we described our method for normalizing proofs to remove some tacticals, in particular the `;` operator. Even as a best-effort process, this significantly increases the number of proofs we can effectively learn from.

Additionally, there are some tactics that all but consume others. For instance, anywhere

that an `auto` would succeed, an `eauto` would also succeed (but not the other way around). The same is true for `constructor` and `econstructor`. In these cases, we can replace all instances of the former with the latter in the training data, resulting in more easily learnable data.

Infrastructure is important Early on in the development process, we started building automated infrastructure for interacting with Coq in a python interface, and scraping proof data. While we thought that this sort of work would be a simple preliminary, we ended up working on this infrastructure throughout the development of Proverbot9001. Because of the complexity of proof systems, and the lack of a consistent API, our interface to these systems had to be constantly evolving as our needs changed, and with any update to the version of Coq that we were using.

At no point during development did we reach an interface that would operate correctly 100% of the time. Instead, we handled common cases, and added more language support as it came up in various benchmarks and use-cases. Even so, our final interface has holes. Until a more consistent interface is implemented, as described in work by Ringer et al [RSSGL20], our interfaces to the Coq proof assistant will likely remain incomplete.

However, there is a direct python API being developed in conjunction with SerAPI that will hopefully someday ease the burden of working with Coq from Python [Ari21]. Additionally, other work [YD19] has successfully used Coq plugins to pull information from Coq directly, which can then be accessed by python in a more uniform format. Future work should consider this approach to ease the maintainence burden, though it does have the downside of being tied to particular Coq versions, sometimes particular branches, whereas the current approach has been shown to operate across several subsequent versions of Coq.

Domain knowledge is key For decades, conventional wisdom has said that machine learning systems benefit from being able to learn their own features, and that attempting to structure them too much leads to ruin, typified by the oft quoted "Every time I fire a linguist, the performance of the speech recognizer goes up" from natural language processing researcher

Frederick Jelinek [JM08].

The first version of Proverbot9001, Proverbot9000, took this idea to an extreme, modeling proofs as simply a sequence of tokens, and always attempting to predict the next token. While this approach was not completely unsuccessful (it successfully predicted "auto" for several prooofs), it lacked the long term memory and structure needed to tackle complicated proofs. Since then, the development of Proverbot9001 has largely been about carefully integrating more domain knowledge into the system.

Since human provers use Coq interactively, basing their actions on the current context feedback, our first improvement was to process the proof context, instead of just the previous proof tokens, to generate the next proof token. Since not all tokens have the same semantic meaning, we seperated prediction of proofs tokens into prediction of tactic names, and prediction of arguments. And since arguments are often highly dependent on the proof context, we created an argument model that takes tokens from the hypotheses names and goal tokens.

Rather than focus purely on developing new ML techniques and models, much of the development of Proverbot9001 has been about understanding Coq usage on a deeper level. Much time was spent running through and studying existing proofs, and referencing the Coq documentation. And in conjunction with this work, we studied users of Coq, and published a paper on this study [RSSGL20].

9.2 Future Work

While Proverbot9001 is the leader in ML for proof search at this time, there is still a long way to go before software verification can be largely automated. Some of that work is simply integrating and expanding approaches from other work, and integerating them into a unified proof search tool. And some is developing new techniques, or adapting techniques from the ML literature that have not yet been applied to proofs.

Integrating proof sequence models As discussed above, the earliest version of Proverbot9001 used a sequence model of proof tokens to generate proofs. As Proverbot9001 was developed, we moved to a system which primarily used the current context to predict each tactic. Though we retained a single integer feature for tracking the last used tactic name.

Concurrent work, TacTok [FBG20], has successfully integrated a sequence model of the previous proof script, with the simpler context encoding system used by ASTactic [YD19]. This work has shown promise, and is able to prove theorems that ASTactic alone cannot solve, even when augmented with the CoqHammer solver-based automation [FBG20]. Though TacTok's experimental results so far are largely subsumed by Proverbot9001, the sequence model has shown enough promise that integrating it into Proverbot9001 would probably improve results.

A combined sequence model would make use of Proverbot9001s action encoding, developed in Section 5.2.2, to encode both tactic and argument tokens, without erasing the distinction between the two. While TacToks model runs over all tactics previously run in the proof script, the combined model would also make use of Proverbot9001s subgoal structure understanding to provide "previous tactics" traces which only track the path to the current subgoal. This will allow it to generalize across subgoals in different positions in a split, and ignore tactics which were applied to a previous subgoal and do not affect the current one.

Tree-based term encodings Proverbot9001 has several systems for encoding terms as input to the predictor. Goal terms are encoded using one different RNN, producing a likelyhood that each token in the goal is the correct argument. And encoded by a differnt RNN as input to the hypothesis argument model. Hypothesis terms are encoded using yet another RNN as part of the argument model, combining with the encoded goal and tactic name to produce a likelyhood that the particular hypothesis is the correct argument.

However, all of these RNN's process terms as linear sequences of tokens, akin to a language model. This ignores the tree structure of the terms, where functions and operators have

child nodes, whose order matters less than their position in the tree. We partially mitigate this by using Coq's `Unset Notation.` mode, which prints all terms as s-expressions, where the operator or function is printed at the beginning of the term. However, this is only a partial fix.

Future versions of Proverbot9001 should integrate a graph- or tree-based neural encoding of terms, to fully take advantage of their structure. This has already been investigated in previous work, like ASTactic [YD19], where TreeLSTM is used to encode the tree structure of the terms. However, neural architectures for processing trees are still in their infancy, and have not been shown to produce significantly better results than token streams yet. As tree-based models progress, proof search tools like Proverbot9001 can integrate the latest developments.

Existential variable handling across goals One of the primary components of Proverbot9001 is the linearizer, which takes proofs using the semicolon tactical (`;`), and turns them into linear proofs which do not make use of that tactical. The semicolon tactical allows proofs to operate on several goals simultaniously; the linearizer attempts to turn this into a proof which completes each subgoal before moving on to the next. While naïvely, this would always be possible, in practice this process is best-effort, and not all proofs are successfully linearized.

This is because branches of a proof are not completely independent; instead, they share mappings of existential variables, which can be an essential aspect of proof completion. Existential variables are holes in terms which must be filled by a term of the right type, though the value of that term is unknown. Often, only a certain value of an existential variable will allow a proof to go through. However, constraints on that existential variable may be different across different branches of the proof where it appears (if they are non-overlapping, then the proof will be non solvable). This means that attempting to resolve the variable in one branch of the proof may be an under-constrained problem, or produce a term which does not satisfy another branches constraints, while resolving it first in another branch produces the correct value.

This suggests two ways to improve Proverbot9001.

First, the linearizer could be improved to detect when an existential variable needs to be first resolved in a future goal, and produce tactics to explicitly switch to that goal, allowing the proof to still be mostly linear, and making the variable resolution more explicit. This would allow us to linearize more proofs than could previously be linearized, and learn from more data.

Second, once this goal switching was made explicit in the training data, Proverbot9001 could be augmented to learn to switch goals when necessary in it's own proofs. In such a system, when an existential variable appears across multiple subgoals, Proverbot9001 would learn a function which selects which goal would most easily find the value of the existential variable. This would allow it to complete more proofs that include existential variables successfully.

Term arguments The biggest hold in Proverbot9001s expressivity is its inability to produce complex tactic arguments. These arguments can range from a simple function application, like in `destruct (andb x y)`, to new lemma statements like those given to the `assert` tactic. We found that Proverbot9001 can still solve a significant number of proofs without access to such tactic arguments, even when the original proof made use of these types of tactic arguments. However, some proofs still cannot be solved without using term arguments. Future work can make use of existing research on neural program synthesis to train a model to produce tree-structured tactic arguments. Such a model could be integrated into Proverbot9001's infrastructure to produce a proof search tool which can solve even more proofs, and train on more data.

Multiple arguments In addition to taking complex term and term-like arguments, some tactics take multiple arguments. Our approach so far has been to remove tactics which use multiple arguments from our training data set, and to not attempt to predict them.

In some cases, like `intros`, tactics with multiple arguments can be replaced statically with a sequence of single-argument tactics. In others, like `subst`, or `auto with`, they cannot. Future work should desugar the former cases to simpler tactics, but attempt to learn the latter case directly. It's not yet clear how such models should be structured; should they be sequence

models, continually producing a next argument until a special end-token, or evaluate each possible argument seperately and use all of those that reach a certain score?

Lemma prediction Proverbot9001 and other tools have been working on the problem of generating a valid proof script from a lemma statement (and environment). However, this is not the only challenge facing software verification; it may not even be the most difficult one. Successful proof development depends on good *proof architecture*; primarily, the breaking down of top level correctness statements into the data structures and intermediary lemmas necessary to prove it.

Future proof synthesis tools will generate not just a proof script, but new lemma statements and supporting data structures for proving them. Parts of this problem are closely related to loop- and data structure-invariant synthesis, a well studied problem [PIS+16, FWSS19, MPMW20, PMNS19, PSM16]; however, this problem is only beginning to be explored in a proof context. In addition to generating lemma statements, these future proof tools will need to generate new inductive definitions for propositions. This part of the problem is more closely related to work in data structure synthesis, which has been studied extensively since the 1970's [Ear73, Ear75, LET18].

Beyond the goal of producing lemmas and data structures which help automated tools verify software, future work could also look at attempting to make these elements interpretable by a user, including having readable names, which has been recently attempted [NPLG20].

Predictions as human guidance Even the best proof search tools will likely never completely subsume human expertise. But automation and human expertise are not mutually exclusive. In the Coq system in an interactive setting, both the human and the automation work together to find a proof term. Humans choose the tactics to invoke, and the automation uses those tactics to search for terms, providing further feedback to the human.

In the future, tools for tactic-script synthesis, like Proverbot9001, can be repurposed to

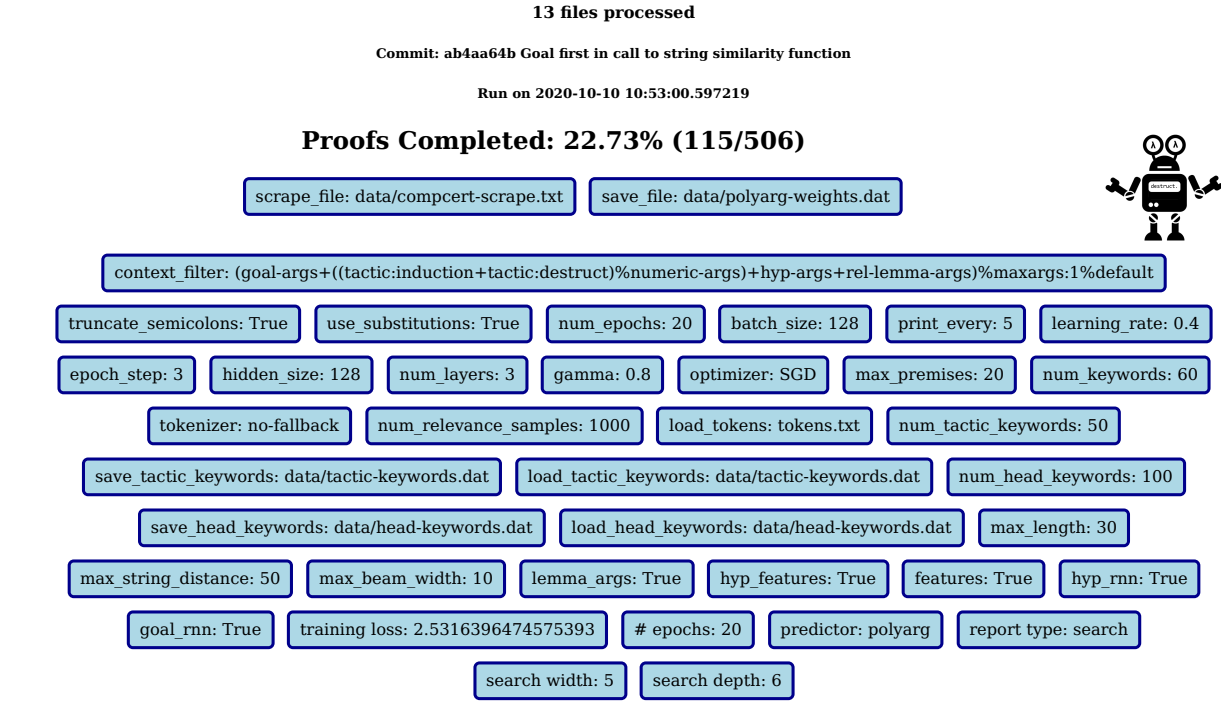
provide guidance to a human prover. The PeaCoq project has already explored providing further interactive guidance to human provers, running multiple possible next steps and displaying the resulting contexts [Rob]. However, this work only ranked possible tactics with a simple, mostly static ranking system. Integrating a learned tactic predictor, like Proverbot9001, into such a system, would greatly increase it's usability.

Appendix A

HTML Reports

When Proverbot9001's main entry point for search is run on files that have been scraped, it generates an HTML report of the result. The report is composed of an index page (see Figure A.1) and multiple details pages (see Figure A.2). The index page has three main parts: overall stats, individual file stats, and parameters/environment.

First, the overall stats report the total number of proofs, the number solved, and the percentage solved. Next, a table reports the same stats per-file, but also breaks down the failed proofs into incomplete (hit a depth limit) or failed (ran out of options to search). Finally, both at the top and bottom of the report, there is metadata information for reproducing the results. This includes the date and time at which it was run, and the repo commit hash used. It also includes parameter values for a variety of values, and at the bottom of the page, the exact commands use to run the report and train the weights.



Filename	Number of Proofs in File	% Proofs Completed	% Proofs Incomplete	% Proofs Failed	Details
common/Globalenvs.v	112	20.54	79.46	0.00	Details
lib/Parmov.v	89	26.97	66.29	6.74	Details
x86/SelectOpproof.v	67	19.40	80.60	0.00	Details
backend/Selectionproof.v	60	15.00	85.00	0.00	Details
flocq/Core/Zaux.v	57	26.32	68.42	5.26	Details
flocq/Calc/Round.v	40	2.50	97.50	0.00	Details
backend/Locations.v	31	64.52	35.48	0.00	Details
cfrontend/Cop.v	25	28.00	72.00	0.00	Details
backend/RTL.v	9	11.11	88.89	0.00	Details
MenhirLib/Validator_complete.v	7	14.29	85.71	0.00	Details
flocq/Prop/Mult_error.v	7	0.00	100.00	0.00	Details
lib/Wfsimpl.v	2	50.00	50.00	0.00	Details
Total	506	22.73	75.49	1.78	

Trained as: ['./src/proverbot9001.py', 'train', 'polyarg', 'data/compcert-scrape.txt', 'data/polyarg-weights.dat', '--load-tokens=tokens.txt', '--context-filter=(goal-args+((tactic:induction+tactic:destruct)%numeric-args)+hyp-args+rel-lemma-args)%maxargs:1%default']
Reported as: ['src/search_file.py', '-P', '--weightsfile=data/polyarg-weights.dat', '--search-width=5', '--search-depth=6', '--output=search-report-test2', '--prelude=CompCert', '-j', '8', './backend/Locations.v', './backend/RTL.v', './backend/Selectionproof.v', './cfrontend/Cop.v', './exportclight/Clightdefs.v', './MenhirLib/Validator_complete.v', './x86/SelectOpproof.v', './flocq/Calc/Round.v', './flocq/Prop/Mult_error.v', './flocq/Core/Zaux.v', './lib/Parmov.v', './lib/Wfsimpl.v', './common/Globalenvs.v']

Figure A.1: The index page of the HTML report. Includes commit/time information, parameter values, and the invocation command, for maximum reproducability. The main table shows statistics about each file, and links to their details pages.

```
 (* src -> dst *)

Definition srcs (m: moves) := List.map (@fst reg reg) m.
Definition dests (m: moves) := List.map (@snd reg reg) m.
(** ** Semantics of moves *)

(** The dynamic  semantics of moves is given in terms of environments.
    An environment is a mapping of registers to their current values. *)

Variable val: Type.
Definition env := reg -> val.
Lemma env_ext:                                                          +
  forall (e1 e2: env),
  (forall r, e1 r = e2 r) -> e1 = e2.
(** The main operation over environments is update: it assigns
  a value [v] to a register [r] and preserves the values of other
  registers. *)

Definition update (r: reg) (v: val) (e: env) : env :=
  fun r' => if reg_eq r' r then v else e r'.
Lemma update_s:                                                         +
  forall r v e, update r v e r = v.
Lemma update_o:                                                         +
  forall r v e r', r' <> r -> update r v e r' =  e r'.
Lemma update_ident:                                                     +
  forall r e, update r (e r) e = e.
Lemma update_commut:                                                    +
  forall r1 v1 r2 v2 e,
  r1 <> r2 ->
  update r1 v1 (update r2 v2 e) = update r2 v2 (update r1 v1 e).
Lemma update_twice:                                                     +
  forall r v e,
  update r v (update r v e) = update r v e.
(** A list of moves [(src1, dst1), ..., (srcN, dstN)] can be given
  three different semantics, as environment transformers.
```

Figure A.2: A portion of the details page for the `lib/Parmov.v` file in CompCert. Each recognized lemma is given a collapsible box, colored by the status of the search; here, `env_ext` is colored red because the search dead-ended before finding a solution, while `update_s` is colored green for success.

Bibliography

[ABDS17] Miltiadis Allamanis, Earl T. Barr, Premkumar T. Devanbu, and Charles A. Sutton. A survey of machine learning for big code and naturalness. *CoRR*, abs/1709.06182, 2017.

[ACI+16] Alexander A. Alemi, François Chollet, Geoffrey Irving, Christian Szegedy, and Josef Urban. Deepmath - deep sequence models for premise selection. *CoRR*, abs/1606.04442, 2016.

[App15] Andrew W. Appel. Verification of a cryptographic primitive: Sha-256. *ACM Trans. Program. Lang. Syst.*, 37(2):7:1–7:31, April 2015.

[Ari21] Emilio Jesús Gallego Arias. Preliminary python api. `https://github.com/charlespwd/project-title`, 2021.

[BBV18] Mislav Balunoviundefined, Pavol Bielik, and Martin Vechev. Learning to solve smt formulas. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 10338–10349, Red Hook, NY, USA, 2018. Curran Associates Inc.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BLR+19] Kshitij Bansal, Sarah M. Loos, Markus N. Rabe, Christian Szegedy, and Stewart Wilcox. Holist: An environment for machine learning of higher-order theorem proving (extended version). *CoRR*, abs/1904.03241, 2019.

[BRV16] Pavol Bielik, Veselin Raychev, and Martin Vechev. Phog: Probabilistic model for code. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2933–2942, New York, New York, USA, 20–22 Jun 2016. PMLR.

[BUG20] Lasse Blaauwbroek, Josef Urban, and Herman Geuvers. Tactic learning and proving for the coq proof assistant. *CoRR*, abs/2003.09140, 2020.

[CCK+17] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 270–286, New York, NY, USA, 2017. ACM.

[Chl13] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. The MIT Press, 2013.

[CK18] Łukasz Czajka and Cezary Kaliszyk. Hammer for coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1):423–453, Jun 2018.

[CvMG+14] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *CoRR*, abs/1406.1078, 2014.

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[DTP16] Hoa Khanh Dam, Truyen Tran, and Trang Pham. A deep language model for software code. *CoRR*, abs/1608.02715, 2016.

[Ear73] Jay Earley. Relational level data structures for programming languages. *Acta Informatica*, 2(4):293–309, Dec 1973.

[Ear75] Jay Earley. High level iterators and a method for automatically designing data structure representation. *Computer Languages*, 1(4):321–342, 1975.

[FBG20] Emily First, Yuriy Brun, and Arjun Guha. Tactok: Semantics-aware proof synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[FHB+20] Jean-Christophe Filliâtre, Hugo Herbelin, Bruno Barras, Bruno Barras, Samuel Boutin, Eduardo Giménez, Samuel Boutin, Gérard Huet, César Muñoz, Cristina Cornes, Cristina Cornes, Judicaël Courant, Judicael Courant, Chetan Murthy, Chetan Murthy, Catherine Parent, Catherine Parent, Christine Paulin-mohring, Christine Paulin-mohring, Amokrane Saibi, Amokrane Saibi, Benjamin Werner, and Benjamin Werner. The coq proof assistant - reference manual version 8.11. Technical report, 2020.

[FOWZ16] Jonathan Frankle, Peter-Michael Osera, David Walker, and S Zdancewic. Example-directed synthesis: a type-theoretic interpretation. *ACM SIGPLAN Notices*, 51:802–815, 01 2016.

[FWSS19] Yotam M. Y. Feldman, James R. Wilcox, Sharon Shoham, and Mooly Sagiv. Inferring inductive invariants from phase structures. *CoRR*, abs/1905.07739, 2019.

[GKKP13] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. *PLDI 2013*, pages 12. 27–38, 2013.

[GKU17] Thibault Gauthier, Cezary Kaliszyk, and Josef Urban. Tactictoe: Learning to reason with hol4 tactics. In Thomas Eiter and David Sands, editors, *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 46 of *EPiC Series in Computing*, pages 125–143. EasyChair, 2017.

[GNMR16] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. Learning invariants using decision trees and implication counterexamples. *SIGPLAN Not.*, 51(1):499–512, January 2016.

[Gul10] Sumit Gulwani. Dimensions in program synthesis. In *PPDP '10 Hagenberg, Austria*, January 2010.

[HDSS18] Daniel Huang, Prafulla Dhariwal, Dawn Song, and Ilya Sutskever. Gamepad: A learning environment for theorem proving. *CoRR*, abs/1806.00608, 2018.

[HK14] Jónathan Heras and Ekaterina Komendantskaya. Acl2(ml): Machine-learning for ACL2. In *Proceedings Twelfth International Workshop on the ACL2 Theorem Prover and its Applications, Vienna, Austria, 12-13th July 2014.*, pages 61–75, 2014.

[HVP+17] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.

[JM08] Daniel Jurafsky and James Martin. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, volume 2. 02 2008.

[KBP13] Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32(11):1238–1274, 2013.

[KBW+18] Daniel Kästner, Joerg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. Compcert: Practical experience on integrating and qualifying a formally verified optimizing compiler. 01 2018.

[KCS17] Cezary Kaliszyk, François Chollet, and Christian Szegedy. Holstep: A machine learning dataset for higher-order logic theorem proving. *CoRR*, abs/1703.00426, 2017.

[KEH+09] 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. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM.

[KHG12] Ekaterina Komendantskaya, Jónathan Heras, and Gudmund Grov. Machine learning in proof general: Interfacing interfaces. *Electronic Proceedings in Theoretical Computer Science*, 118, 12 2012.

[KL12] Ekaterina Komendantskaya and Kacper Lichota. Neural networks for proof-pattern recognition. volume 7553, pages 427–434, 09 2012.

[KUMO18] Cezary Kaliszyk, Josef Urban, Henryk Michalewski, and Mirek Olsák. Reinforcement learning of theorem proving. *CoRR*, abs/1805.07563, 2018.

[KV13] Laura Kovács and Andrei Voronkov. First-order theorem proving and vampire. volume 8044, pages 1–35, 07 2013.

[LAR17] Fan Long, Peter Amidon, and Martin Rinard. Automatic inference of code transforms for patch generation. pages 727–739, 08 2017.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[LET18] Calvin Loncaric, Michael D Ernst, and Emina Torlak. Generalized data structure synthesis. In *Proceedings of the 40th International Conference on Software Engineering*, pages 958–968, 2018.

[LISK17] Sarah M. Loos, Geoffrey Irving, Christian Szegedy, and Cezary Kaliszyk. Deep network guided proof search. *CoRR*, abs/1701.06972, 2017.

[MBM+16] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Tim Harley, Timothy P. Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 1928–1937. JMLR.org, 2016.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[MLJ+14] Lili Mou, Ge Li, Zhi Jin, Lu Zhang, and Tao Wang. TBCNN: A tree-based convolutional neural network for programming language processing. *CoRR*, abs/1409.5718, 2014.

[MMSW10] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 237–248, New York, NY, USA, 2010. ACM.

[MPMW20] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. Data-driven inference of representation invariants. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 1–15, New York, NY, USA, 2020. Association for Computing Machinery.

[NPLG20] Pengyu Nie, Karl Palmskog, Junyi Jessy Li, and Milos Gligoric. Deep generation of Coq lemma names using elaborated terms. In *International Joint Conference on Automated Reasoning*, pages 97–118, 2020.

[OZ15] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *SIGPLAN Not.*, 50(6):619–630, June 2015.

[Pau93] Lawrence C. Paulson. Natural deduction as higher-order resolution. *CoRR*, cs.LO/9301104, 1993.

[PIS+16] Oded Padon, Neil Immerman, Sharon Shoham, Aleksandr Karbyshev, and Mooly Sagiv. Decidability of inferring inductive invariants. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, page 217–231, New York, NY, USA, 2016. Association for Computing Machinery.

[PLR+19] Aditya Paliwal, Sarah M. Loos, Markus N. Rabe, Kshitij Bansal, and Christian Szegedy. Graph representations for higher-order logic and theorem proving. *CoRR*, abs/1905.10006, 2019.

[PMNS19] Saswat Padhi, Todd Millstein, Aditya Nori, and Rahul Sharma. Overfitting in synthesis: Theory and practice. In *Computer Aided Verification (CAV)*, July 2019.

[PSM16] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *SIGPLAN Not.*, 51(6):42–56, June 2016.

[Rob] Valentin Robert. *Front-end tooling for building and maintaining dependently-typed functional programs*. PhD thesis.

[RSSGL20] Talia Ringer, Alex Sanchez-Stern, Dan Grossman, and Sorin Lerner. Replica: Repl instrumentation for coq analysis. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, New York, NY, USA, 2020. Association for Computing Machinery.

[Sch13] Stephan Schulz. System Description: E 1.8. In Ken McMillan, Aart Middeldorp, and Andrei Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[SIS17] Taro Sekiyama, Akifumi Imanishi, and Kohei Suenaga. Towards proof synthesis guided by neural machine translation for intuitionistic propositional logic. *CoRR*, abs/1706.06462, 2017.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, Oct 2017.

[Tan11] O. Tange. Gnu parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, Feb 2011.

[WWP+15] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 357–368, New York, NY, USA, 2015. ACM.

[YCER11] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *PLDI*, 2011.

[YD19] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. *CoRR*, abs/1905.09381, 2019.

[ZCM+19] Zsolt Zombori, Adrián Csiszárik, Henryk Michalewski, Cezary Kaliszyk, and Josef Urban. Curriculum learning and theorem proving. *Conference on Artificial Intelligence and Theorem Proving*, 2019.