mCoq: Mutation Analysis for Coq Verification Projects

Kush Jain
kjain14@utexas.edu
The University of Texas at Austin

Karl Palmskog
palmskog@acm.org
KTH Royal Institute of Technology

Ahmet Celik
celik@fb.com
Facebook, Inc.

Emilio Jesús Gallego Arias
e@x80.org
Équipe πr^2, Inria, IRIF, Univ. de Paris

Milos Gligoric
gligoric@utexas.edu
The University of Texas at Austin

ABSTRACT

Software developed and verified using proof assistants, such as Coq, can provide trustworthiness beyond that of software developed using traditional programming languages and testing practices. However, guarantees from formal verification are only as good as the underlying definitions and specification properties. If properties are incomplete, flaws in definitions may not be captured during verification, which can lead to unexpected system behavior and failures. Mutation analysis is a general technique for evaluating specifications for adequacy and completeness, based on making small-scale changes to systems and observing the results. We demonstrate mCoq, the first mutation analysis tool for Coq projects. mCoq changes Coq definitions, with each change producing a modified project version, called a mutant, whose proofs are exhaustively checked. If checking succeeds, i.e., the mutant is live, this may indicate specification incompleteness. Since proof checking can take a long time, we optimized mCoq to perform incremental and parallel processing of mutants. By applying mCoq to popular Coq libraries, we found several instances of incomplete and missing specifications manifested as live mutants. We believe mCoq can be useful to proof engineers and researchers for analyzing software verification projects. The demo video for mCoq can be viewed at: https://youtu.be/QhigpfQ7dNo.

CCS CONCEPTS

• **Theory of computation** → *Logic and verification*; • **Software and its engineering** → *Software verification and validation.*

KEYWORDS

Mutation analysis, Coq, proof assistants, deductive verification

ACM Reference Format:
Kush Jain, Karl Palmskog, Ahmet Celik, Emilio Jesús Gallego Arias, and Milos Gligoric. 2020. mCoq: Mutation Analysis for Coq Verification Projects. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3382156

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7122-3/20/05.
https://doi.org/10.1145/3377812.3382156

1 INTRODUCTION

Software developed and formally verified using proof assistants, such as Coq, is significantly more trustworthy than software written in traditional programming languages, such as Java. Large programs verified in Coq include the CompCert C compiler [14], which has recently found applications in embedded systems [12]. However, guarantees from formal verification are only as good as the underlying definitions and properties (specifications). Verified properties may be incomplete, which can lead to unexpected system behavior and even bugs at runtime [7]. Mutation analysis is a general technique for evaluating specifications for adequacy and completeness, and is based on making small-scale changes to code and observing whether verification or testing succeeds or fails [1, 9].

We demonstrate mCoq, the first mutation analysis tool for Coq projects. mCoq applies a set of *mutation operators* to Coq definitions, with each successful application generating a modified version (*mutant*) of the project. If all proofs are successfully checked, the mutant is declared *live*; otherwise, the mutant is declared *killed*. As in mutation testing, live mutants may indicate incomplete specifications, e.g., properties that are sometimes vacuously true [2, 18].

mCoq is implemented in OCaml, Java, and Python, and supports projects that use Coq version 8.10. Source code of all mCoq components is publicly available. Our OCaml code was integrated into the official releases of Coq and SerAPI and is available as part of those projects; source code for the other components can be obtained from: https://cozy.ece.utexas.edu/mcoq

We evaluated an earlier version of mCoq on 12 medium and large scale Coq projects, finding several incomplete specifications in widely used libraries [3]. However, the earlier version of mCoq was closely tied to our evaluation infrastructure and relied on a modified version of Coq. In contrast, the version of mCoq presented here has been adapted and extended for general use and works with unmodified Coq version 8.10 and SerAPI version 0.7. In addition, mCoq can now generate detailed HTML reports that graphically pinpoint all live and killed mutants in the Coq code.

We believe mCoq can be useful to proof engineers and researchers for analyzing software verification projects and for evaluating proof engineering techniques [19].

2 TECHNIQUE AND IMPLEMENTATION

In this section, we briefly describe our mutation analysis technique, which we call *mutation proving* [3], that mCoq implements. We then explain the main mCoq components and workflow.

Mutation proving is based on exhaustively applying *mutation operators* to Coq projects. When an operator is successfully applied

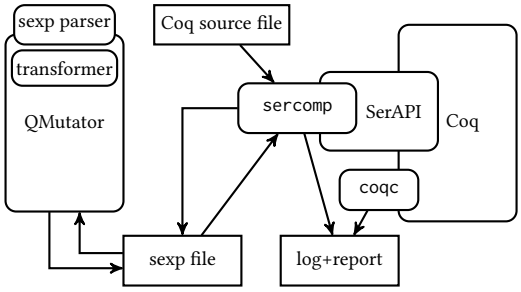


Figure 1: mCoq components, inputs, and outputs.

to a source file, it generates a mutant where a Coq definition is different from before. For example, an operator can change addition of natural numbers to subtraction, or remove the head of a list. Note that in analogy with mutation testing, which does not mutate tests, we do not change specification properties or proofs. If all proofs associated with a mutant are successfully checked, the mutant is reported live; otherwise, it is reported killed. A project's *mutation score* is defined as the percentage of killed mutants out of all syntactically distinct generated mutants. A low mutation score intuitively, but not unambiguously, indicates that specifications may be incomplete or missing for some definitions.

mCoq implements mutation proving by first serializing all Coq code to S-expressions [15] (sexps) using the SerAPI library, then transforming the sexps pinpointed by mutation operators, and finally deserializing the results and checking all associated proofs. Fig. 1 shows the key components of mCoq, which are essentially unchanged from the initial version. The `sercomp` command-line tool, which we developed and is included in SerAPI, handles (de)serialization. The standard `coqc` tool proof-checks source files. Finally, the QMutator tool that we wrote in Java transforms the sexps.

Checking a Coq proof can be time-consuming, e.g., due to invocations of arithmetic constraint solvers in proof scripts. Our optimized mode for mCoq performs incremental analysis that avoids unnecessary (de)serialization and parallelizes checking of each individual mutant. In the evaluation of our previous tool version, we found that this mode consistently outperforms other parallel modes. We also include a basic mode, called Default, which naively (de)serializes and checks all files for every mutant.

To collect mutant data for reporting, we process the log files generated by QMutator using Python, computing information on killed and live mutants, mutation scores, and execution time. We then generate HTML reports in a format inspired by the format of the JaCoCo library [11], as explained in more detail in Section 3.

3 TOOL INSTALLATION AND USAGE

This section describes how users can install and integrate mCoq into their workflows.

3.1 Installation

The first installation step is to install Coq 8.10 and SerAPI 0.7. We recommend installing these dependencies via the OCaml-based package manager OPAM [17], version 2.0.5 or later:

```
$ opam update
$ opam install coq.8.10.2 coq-serapi.8.10.0+0.7.0
```

Table 1: Command-line Arguments Available in mCoq.

Argument	Description
`project`	name of project to run mCoq on
`sha`	SHA of project to run mCoq on
`buildcmd`	Coq build command for the library
`url`	URL to clone the project from via Git
`rdir`	`-R` option for `sercomp`
`qdir`	`-Q` option for `sercomp`
`skipeq`	enable/disable detection of equivalent mutants
`nocheck`	skip checking for mCoq dependencies
`dry`	only print proof checking commands for each mutant
`mutator`	mode to run mCoq in
`mutations`	comma separated list of all mutations to run mCoq on (if omitted, all mutations are run)
`skipreport`	skip generating the report
`skipmutations`	only generate the report (if the log is available from a prior run)
`report_dir`	directory for storing the final report
`threads`	number of parallel threads

Then, users should clone the mCoq repository on GitHub:

```
$ git clone https://github.com/EngineeringSoftware/mcoq.git
$ cd mcoq && git checkout v1.0
```

Additionally, we assume JDK 8 or later and the Gradle build system [8] are installed. Finally, our entry script (`mcoq.py`) for the tool requires Python 3 to be available on the system.

3.2 Usage

After installation, users can interact with mCoq via `mcoq.py`, e.g.,

```
$ ./mcoq.py --help
```

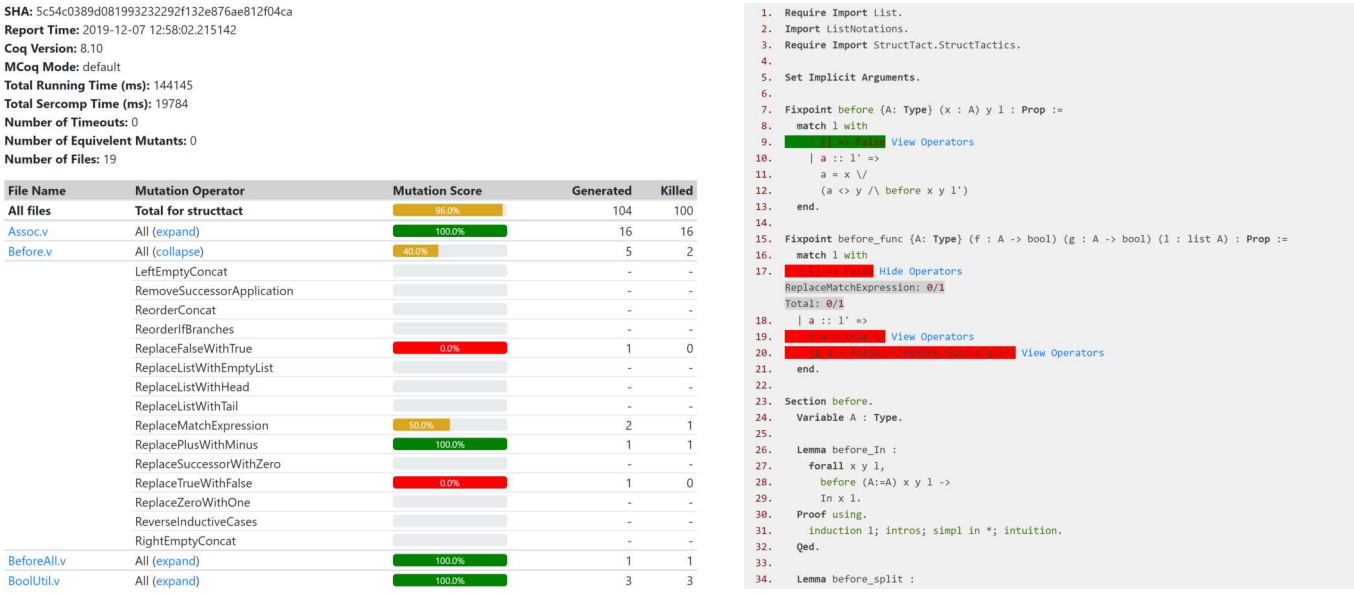
Applying mCoq to a Coq project requires that the project code (1) lives in a Git-based repository and (2) contains a `_CoqProject` file in the root directory listing all relevant Coq source files, in the format used by the `coq_makefile` tool [5]. If these requirements are met, no changes are required to use mCoq on a project.

For example, running the following command in the `mcoq` directory fetches the Coq project StructTact at revision 82a85b7 from its GitHub repository and mutates it using two parallel threads:

```
$ ./mcoq.py --project StructTact --sha 82a85b7 --threads 2 \
    --url https://github.com/uwplse/StructTact.git \
    --buildcmd "./configure && make -j2" --qdir ".,StructTact"
```

This command runs mCoq using the default configuration, meaning, e.g., that all mutation operators are enabled. Table 1 lists and describes all arguments currently available for the tool. Note that changing the mode via the option `--mutator` is provided to enable further research on efficient mutation analysis for Coq; we do not expect this option to be used for other purposes.

When the execution of the above command has completed, the result is a log file and an HTML report. The log file contains detailed information about the entire execution (trace), generated mutants, and execution time and outcome (killed or live) for each mutant. The log is used both for debugging the tool (when necessary) and for generating the more user-friendly report. The report is automatically generated in the `reports` directory and can be displayed with any modern web browser. We provide an example report at: https://cozy.ece.utexas.edu/mcoq/report.



SHA: 5c54c0389d081993232292f132e876ae812f04ca
Report Time: 2019-12-07 12:58:02.215142
Coq Version: 8.10
MCoq Mode: default
Total Running Time (ms): 144145
Total Sercomp Time (ms): 19784
Number of Timeouts: 0
Number of Equivelent Mutants: 0
Number of Files: 19

File Name	Mutation Operator	Mutation Score	Generated	Killed
All files	**Total for structtact**	96.0%	104	100
Assoc.v	All (expand)	100.0%	16	16
Before.v	All (collapse)	40.0%	5	2
	LeftEmptyConcat		-	-
	RemoveSuccessorApplication		-	-
	ReorderConcat		-	-
	ReorderIfBranches		-	-
	ReplaceFalseWithTrue	0.0%	1	0
	ReplaceListWithEmptyList		-	-
	ReplaceListWithHead		-	-
	ReplaceListWithTail		-	-
	ReplaceMatchExpression	50.0%	2	1
	ReplacePlusWithMinus	100.0%	1	1
	ReplaceSuccessorWithZero		-	-
	ReplaceTrueWithFalse	0.0%	1	0
	ReplaceZeroWithOne		-	-
	ReverseInductiveCases		-	-
	RightEmptyConcat		-	-
BeforeAll.v	All (expand)	100.0%	1	1
BoolUtil.v	All (expand)	100.0%	3	3

(a) Project-base view

```
1.  Require Import List.
2.  Import ListNotations.
3.  Require Import StructTact.StructTactics.
4.
5.  Set Implicit Arguments.
6.
7.  Fixpoint before {A: Type} (x : A) y l : Prop :=
8.    match l with
9.      [1] => False  View Operators
10.     | a :: l' =>
11.       a = x \/
12.       (a <> y /\ before x y l')
13.   end.
14.
15. Fixpoint before_func {A: Type} (f : A -> bool) (g : A -> bool) (l : list A) : Prop :=
16.   match l with
17.     [1] => False  Hide Operators
      ReplaceMatchExpression: 0/1
      Total: 0/1
18.   | a :: l' =>
19.     f a = true  View Operators
20.     [2] => False && before_func f g l'  View Operators
21.   end.
22.
23. Section before.
24.   Variable A : Type.
25.
26.   Lemma before_In :
27.     forall x y l,
28.       before (A:=A) x y l ->
29.       In x l.
30.   Proof using.
31.     induction l; intros; simpl in *; intuition.
32.   Qed.
33.
34.   Lemma before_split :
```

(b) File-based view

Figure 2: HTML report automatically generated by MCoq for StructTact project used in our evaluation.

3.3 Reports in Detail

Figure 2 shows two screenshots of the HTML report generated for the StructTact project, which is one of the projects we used in our evaluation (see Section 4). To design the report pages generated by MCoq, we initially took inspiration in JaCoCo [11], a popular code coverage tool for Java projects. However, JaCoCo's task is simpler: it only provides a binary flag for each line (a line is colored green if covered and red if not covered). For mutation analysis, we have to report (potentially) multiple mutants per line, the operator that generated each mutant, and the outcome of the analysis.

Figure 2(a) gives an overview of the outcome of the mutation analysis. Specifically, the top part of the figure shows the SHA of the project for which the report was generated, time when the report was generated, Coq version, MCoq mode, total running time (in milliseconds), total time to run `sercomp`, number of mutants that were killed due to timeout, number of syntactically equivalent mutants, and total number of source files in the project. The bottom part of the figure shows the table that includes the key metrics for mutation analysis: mutation score, number of generated mutants, and number of killed mutants. We show these metrics for the entire project in the first row ("All files"). Subsequent rows shows one file per line (sorted alphabetically). A user has the option to expand "Mutation Operator" and see the score per operator; this is shown in the figure for the file `Before.v`. We use green, red, and yellow, for 100%, 0%, and any other mutation score, respectively; gray with no number is used for operators that generated no mutant.

If a user clicks on the file `Before.v`, she sees the page in Figure 2(b) displaying the file contents, including line numbers and syntax highlighting. Additionally, lines that contain at least one generated mutant are highlighted; green color (e.g,. line 9 in the example) is used for those lines where all mutants are killed and red (e.g., line 17 in the example) is used for those lines where at least one mutant is live. If a user would like to see the details about mutants for a specific line, she can click on "View Operators", which will show one line per operator for operators that generated at least one mutant (see lines between 17 and 18 in the example).

4 EVALUATION

To evaluate MCoq, we performed mutation analysis of several popular Coq libraries. We report the number of generated, killed, and live mutants. We also summarize the speedup for our optimized mutation analysis mode compared to the basic mode. Finally, we summarize our findings from manual inspection of live mutants.

The numbers that we report are obtained with the latest version of MCoq compatible with Coq 8.10. However, we also discuss the numbers we obtained with the previous version of the tool compatible with Coq 8.9 [3]; we performed inspection of live mutants found with the previous version of MCoq.

4.1 Coq Projects Under Study

Table 2 lists the name of every project we used in the evaluation, along with the revision SHA we used in the experiments. If SHAs used for the previous MCoq evaluation were not compatible with Coq 8.10, we used more recent SHAs. The table also lists the number of generated, killed, and live mutants, as well as mutation score.

4.2 Summary of Results

We ran all experiments for our evaluation on a 6-core Intel Core i7-8700 CPU @ 3.20GHz machine with 64GB of RAM, running Ubuntu Linux 18.04.1 LTS. We limit the number of parallel threads to be at or below the number of physical CPU cores.

Table 2: Projects Used in the Evaluation.

Project	SHA	#Generated	#Killed	#Live	Score [%]
ATBR	366ac237	355	335	20	94.36
FCSL PCM	b34fce32	115	112	3	97.39
Flocq	7ec13200	416	372	44	89.42
Huffman	50687911	369	366	3	99.18
MathComp	91fa7b57	1076	1060	16	98.51
PrettyParsing	189a2625	282	235	47	83.33
Bin. Rat. Numbers	7b9cc06d	365	352	13	96.43
Quicksort Compl.	0a6eed8b	681	637	44	93.53
Stalmarck	6932ed8a	565	526	39	93.09
Coq-std++	005887ee	583	528	55	90.56
StructTact	82a85b7e	104	100	4	96.15
TLC	4babc16c	400	306	94	76.50
Avg.	n/a	442.58	410.75	31.83	92.37
Total	n/a	5311	4929	382	n/a

The total number of generated, killed, and live mutants are 5311, 4929, and 382, respectively. Mutation score varies between 76.50% (TLC) and 97.39% (FCSL PCM), and is 92.37% on average across all projects. Note that these numbers only differ slightly from our original evaluation [3] due to the update to Coq 8.10 (from Coq 8.9) and updates in the projects (for those projects that we have to move to new SHAs that support Coq 8.10).

Our original evaluation [3] showed that the mode with parallel and incremental checking substantially outperformed all other modes. Thus, we only ran this mode, in addition to the Default mode, in our latest experiment. Our results confirm that the most efficient mode speeds up mutation analysis by 74% over the Default mode. The speedup values are only marginally different from the original evaluation. Finally, we manually inspected 74 live mutants (out of 361 live mutants we found in our earlier evaluation [3]) and determined that 33 indicate *incomplete* specifications, 30 indicate *omitted* specifications, and 11 are semantically equivalent.

4.3 Limitations and Future Work

Our design of the initial set of mutation operators was inspired by our extensive experience with Coq and prior work on mutation analysis for functional languages [13]. Designing specialized operators based on the way a project uses libraries is an exciting future direction. MCoq targets only projects written in Coq; more research is needed to design and evaluate an extensive set of mutation operators for other proof assistants, such as Lean [6] and Isabelle/HOL [16]. We also plan to explore other ways to detect semantically equivalent mutants, e.g., convertibility in Coq [4] and further analyze killed mutants [10, 20]. Finally, MCoq currently applies operators on Coq abstract syntax obtained immediately after parsing. An alternative approach is to mutate representations available at the later elaboration phase of type checking in Coq. We plan to explore this direction in the future.

5 CONCLUSION

We demonstrated MCoq, the first tool for mutation analysis of Coq projects, useful for detecting incomplete specifications. MCoq takes a Coq project as an input, generates mutants, finds what mutants are killed (or live), and generates a report. Our original modifications to Coq and SerAPI, needed to perform mutation analysis, have been accepted by its developers into the latest versions of these projects. This enables a smooth installation and integration of MCoq into existing workflows. Considerable care has been taken to ensure smooth evolution of the functionality in Coq and SerAPI which MCoq relies on; we expect only minor effort to maintain MCoq over time as Coq itself evolves. For our evaluation, we used MCoq to perform mutation analysis on several popular Coq libraries and found many incomplete specifications. We believe that MCoq is ready for a wider use in software verification projects by both proof engineers and researchers.

ACKNOWLEDGMENTS

We thank Marinela Parovic for earlier contributions to the project. We also thank Arthur Charguéraud, Georges Gonthier, Farah Hariri, Catalin Hritcu, Robbert Krebbers, Pengyu Nie, Zachary Tatlock, James R. Wilcox and Théo Zimmermann for their feedback on this work. This work was partially supported by the US National Science Foundation under Grant No. CCF-1652517.

REFERENCES

[1] Paul Ammann and Jeff Offutt. 2008. *Introduction to Software Testing*. Cambridge University Press.
[2] Thomas Ball and Orna Kupferman. 2008. Vacuity in Testing. In *Tests and Proofs*. 4–17.
[3] Ahmet Celik, Karl Palmskog, Marinela Parovic, Emilio Jesús Gallego Arias, and Milos Gligoric. 2019. Mutation Analysis for Coq. In *International Conference on Automated Software Engineering*. 539–551.
[4] Coq Development Team. 2019. Coq Manual: Conversion rules. https://coq.inria.fr/distrib/V8.10.2/refman/language/cic.html.
[5] Coq Development Team. 2019. Coq Manual: Utilities. https://coq.inria.fr/distrib/V8.10.2/refman/practical-tools/utilities.html.
[6] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 2015. The Lean Theorem Prover (System Description). In *International Conference on Automated Deduction*. 378–388.
[7] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. 2017. An Empirical Study on the Correctness of Formally Verified Distributed Systems. In *European Conference on Computer Systems*. 328–343.
[8] Gradle Team. 2019. Gradle. https://gradle.org.
[9] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E. McKenney, and Josie Holmes. 2018. How Verified (or Tested) is My Code? Falsification-driven Verification and Testing. *Automated Software Engineering* 25, 4 (2018), 917–960.
[10] Catalin Hritcu, John Hughes, Benjamin C. Pierce, Antal Spector-Zabusky, Dimitrios Vytiniotis, Arthur Azevedo de Amorim, and Leonidas Lampropoulos. 2013. Testing Noninterference, Quickly. In *International Conference on Functional Programming*. 455–468.
[11] JaCoCo Development Team. 2019. JaCoCo Java Code Coverage Library. https://www.eclemma.org/jacoco.
[12] Daniel Kästner, Ulrich Wünsche, Jörg Barrho, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. 2018. CompCert: Practical Experience on Integrating and Qualifying a Formally Verified Optimizing Compiler. In *European Congress on Embedded Real Time Software and Systems*. 1–9.
[13] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. 2014. MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs. In *International Symposium on Software Testing and Analysis*. 429–432.
[14] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115.
[15] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195.
[16] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer.
[17] OPAM Team. 2019. OCaml Package Manager. https://opam.ocaml.org.
[18] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* 112 (2019), 275–378.
[19] Talia Ringer, Karl Palmskog, Ilya Sergey, Milos Gligoric, and Zachary Tatlock. 2019. QED at Large: A Survey of Engineering of Formally Verified Software. (2019), 102–281.
[20] Talia Ringer, Nathaniel Yazdani, John Leo, and Dan Grossman. 2018. Adapting Proof Automation to Adapt Proofs. In *Certified Programs and Proofs*. 115–129.