A Regression Proof Selection Tool For Coq

Ahmet Celik, Karl Palmskog, and Milos Gligoric
The University of Texas at Austin, University of Illinois at Urbana-Champaign, The University of Texas at Austin
ahmetcelik@utexas.edu,palmskog@illinois.edu,gligoric@utexas.edu

ABSTRACT

Large-scale software verification projects increasingly rely on proof assistants, such as Coq, to construct formal proofs of program correctness. However, such proofs must be checked after every change to a project to ensure expected program behavior. This process of regression proving can require substantial machine time, which is detrimental to productivity and trust in evolving projects. We present ICoq, the first regression proof selection tool. ICoq tracks fine-grained dependencies between Coq definitions, propositions, and proofs, and only checks those proofs affected by changes between two revisions. ICoq is suitable for workflows involving version control and continuous integration services, e.g., Travis CI. We applied ICoq to track dependencies across many revisions in several large Coq projects and measured the time savings compared to proof checking from scratch and when using Coq's timestamp-based toolchain for incremental checking. Our results show that proof checking with ICoq is up to 10 times faster than the former and up to 3 times faster than the latter. The demo video for ICoq can be found at: https://www.youtube.com/watch?v=egFnHkH5pXI.

CCS CONCEPTS

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

KEYWORDS

Proof assistants, regression proof selection, proof engineering, Coq

ACM Reference Format:
Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2018. A Regression Proof Selection Tool For Coq. In *ICSE '18 Companion: 40th International Conference on Software Engineering , May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183440.3183493

1 INTRODUCTION

Software verification projects that construct formal proofs of program correctness using proof assistants, such as Coq [4], are reaching an unprecedented scale. Representative large projects target critical domains, e.g., compilers [15], file systems [10], and distributed systems [21]. Such projects accrue tens of person-years of effort and hundreds of thousands of lines of code (including proofs).

Typically, engineers write programs in Coq's purely functional, strongly-typed programming language, and then build formal proofs in Coq of key program properties using procedures called *tactics*. Certified programs are then *extracted* to a practical programming language such as OCaml and integrated into larger software systems. However, when engineers revise or extend Coq programs, previously proven properties must be *reestablished* to trust extracted code, since even small changes can break a proof. Due to the use of demanding tactics, e.g., arithmetic constraint solvers, and a growing number of proofs, this process of *regression proving* can require considerable machine time, and therefore negatively impact engineers' productivity and trust in evolving large-scale projects [5].

In previous work, we suggested an analogy between *proofs* and *tests*, specifically, that establishing a proof of a program property is analogous to running many (possibly an infinite number of) program tests. Intuitively, regression proving corresponds to running a test suite to check that a revised program is error free. Using this analogy, we proposed a technique for *regression proof selection* [9], i.e., for checking only those proofs affected by a change to a project, mirroring earlier techniques for *regression test selection* in traditional software development [17].

We demonstrate ICoq, the first tool that implements regression proof selection. ICoq works by tracking dependencies between Coq definitions, propositions, and proofs. When presented with a set of changes to Coq files, ICoq uses this knowledge of dependencies to only check the proofs affected by the changes, potentially saving significant time in comparison to checking everything from scratch. While the initial version of ICoq was tailored for evaluation of our regression proof selection technique [9], the version presented in this paper has been adapted and packaged for general use, and can be directly integrated into projects using version control and continuous integration services (CISs), e.g., Travis CI [13, 18].

ICoq is implemented in OCaml, Java, and bash, and supports projects using Coq version 8.5. Most ICoq components are publicly available in source form; the following URL is the best starting point for obtaining the tool: http://cozy.ece.utexas.edu/icoq.

2 TECHNIQUE AND IMPLEMENTATION

This section briefly describes the technique we recently introduced [9] that ICoq implements, and explains the main ICoq components. Prior to describing ICoq, we outline the proof-checking mechanisms in Coq that enabled our implementation.

2.1 Asynchronous Proof Checking in Coq

Definitions of functions and lemmas processed by Coq are written in the Gallina language, and reside along with proof scripts in files ending in `.v`. The standard Coq batch proof checking ("compilation") tool, `coqc`, takes a `.v` file as input, unconditionally processes it top-down, and produces a binary `.vo` file with all constructs, including proofs. Since files may depend on other files, checking all proofs in a Coq project requires basic dependency analysis. The standard `coq_makefile` tool generates a Makefile which, by default, calls a syntactic analysis tool, called `coqdep`, for this purpose [2].

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3183493

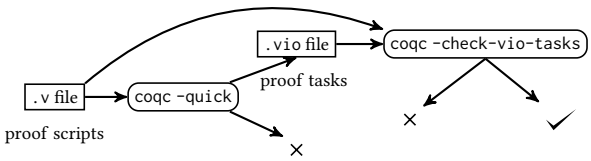


Figure 1: Coq asynchronous proof checking workflow.

To get around the legacy restriction to top-down, file-oriented proof processing, iCoq builds on Coq's recently added asynchronous proof checking capabilities [6]. More precisely, Coq version 8.5 introduced the option to *quick-compile* `.v` files to the binary `.vio` format, a process which avoids checking (and emitting representations of) proofs that have been indicated as *opaque* by ending with `Qed`. Such `.vio` files contain *proof-checking tasks*, which can be performed individually and out-of-order by issuing a `coqc` command referencing the task identifier. A Coq user can depend on more rapidly produced `.vio` files in lieu of `.vo` files for most projects, but must then assume that all proofs are correct, or manually choose to check key proofs. Figure 1 illustrates the workflow with quick compilation and asynchronous proof checking.

Coq uses a notion of *sections* to organize common assumptions made in a collection of lemmas in a `.v` file. Normally, Coq determines which assumptions are used in each lemma only when reaching the end of the section. However, for asynchronous checking, proofs must be annotated up front with the assumptions they use. iCoq assumes that all `.v` files are appropriately annotated, so that proof checking is completely avoided during quick compilation. Annotations can be added automatically to an existing project [3].

2.2 iCoq Phases and Components

iCoq processes Coq projects in three phases (each similar to the corresponding phase in regression test selection tools, e.g., [12, 16, 17]): *analysis*, *proof checking*, and *dependency collection*. In the analysis phase, iCoq detects proofs that are affected by changes made since the last run of iCoq. In the proof checking phase, iCoq checks the proofs selected in the analysis phase (but no other proofs). Finally, in the collection phase, iCoq obtains the new dependencies that will be used in the next run of the tool.

iCoq consists of several independent components; each phase involves one or more of these components. We first describe what the components do and then give details on how they interact during the phases.

`coqdepends` plugin: To extract dependencies from compiled Coq files (`.vo` and `.vio`), we extended prior work on the `coq-dpdgraph` Coq plugin [1], which builds dependency graphs for given identifiers (proofs) or modules (files). In essence, the derived plugin, called `coqdepends`, traverses a term abstract syntax tree (AST), as it is represented in the Coq backend, and records the globally unique *kernel name* of all referenced identifiers it encounters, such as those of lemmas and functions. By performing the dependency extraction at the level of ASTs in compiled files, our tool is isolated from complexities at the Gallina level, such as custom notations.

`coqast` plugin: To compare Coq functions and propositions across project revisions, we developed a plugin for computing short summaries (checksums) of term ASTs that capture the *structure* of the trees. We use a technique for computing checksums based on cryptographic hashes that was shown to be effective at programming language syntax fingerprinting by Chilowicz et al. [11].

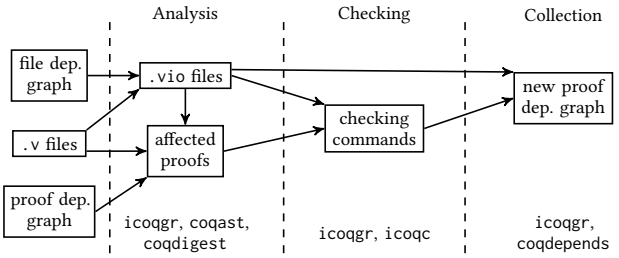


Figure 2: iCoq workflow, with phases and used components.

`coqdigest` tool: Since we cannot compute digests of *proof* ASTs without actually performing all the proof-checking work (that we are trying to avoid), we use digests of the actual proof scripts in the `.v` files. From the standard `coqdoc` tool which translates `.v` files into documentation, we derived a tool dubbed `coqdigest` that extracts proof scripts and returns a checksum of the results.

`icoqc` tool: The AST of a proof for a proof task in a `.vio` file is only available when the proof task completes. Yet, to properly update the proof dependency graph for the next revision, all dependencies must be extracted from such ASTs. Consequently, we extended the `coqc` tool with an additional command that, when given a `.vio` file, its `.v` file, and a proof task, performs the task and then outputs all the dependencies in the resulting proof using the technique from `coqdepends`. Since our `coqc` extension only uses the existing proof checking facilities, it does not affect the soundness of Coq.

`icoqgr` tool: We implemented our own dependency graph builder and dependency analysis in Java. The resulting program reads files (in JSON format) output by the Coq tools and plugins, as well as JSON representations of dependency graphs from previous revisions, and finally writes the updated dependency graphs to disk.

iCoq workflow: Figure 2 shows the high-level iCoq workflow. As indicated in the figure, `icoqgr` is involved in every phase of iCoq. In contrast, `coqast` and `coqdigest` are only involved in the analysis phase of iCoq, while `icoqc` and `coqdepends` are only involved in the checking and collection phases, respectively.

First, for each `.v` in the project, `icoqgr` checks whether its checksum is still the same since the last run of iCoq. Then, `icoqgr` runs `coqdep` on changed files and builds an up-to-date file dependency graph that includes checksums. This graph is used to quick-compile all affected `.v` files into `.vio` files. `coqast` then processes the `.vio` files and obtains checksums for ASTs of all non-opaque Coq identifiers. Independently, `coqdigest` determines the proof tasks available in each changed `.v` file, and obtains the checksum of each proof script associated with a proof task. Based on the obtained proof tasks and checksums, `icoqgr` builds an updated identifier dependency graph from the old graph, where each modified identifier is marked. By traversing marked identifiers in the graph, `icoqgr` marks all transitively affected identifiers, yielding a set of affected proofs and their associated proof tasks. When these proof tasks are run, `icoqc` collects each proof's dependencies; `coqdepends` collects the dependencies of every other changed identifier. `icoqgr` then writes a fully up-to-date identifier dependency graph to disk for use in future runs of iCoq.

3 USAGE

This section describes the steps that a user has to take to enable using iCoq for a Coq project (that typically resides in a version-controlled repository). We describe requirements, installation procedure, and integration of iCoq into a common workflow. We recommend that iCoq is used as a complete replacement for `coq_makefile` to perform batch proof checking and code extraction in a project, either on the user's own machine or in a CIS.

3.1 Requirements and Installation

iCoq requires the following software to be installed: bash, Java 8 or later, and OPAM 1.2.2 with OCaml 4.02.3 or later. Additionally, to use iCoq, a project's `.v` files must be compatible with Coq 8.5 and be annotated with the assumptions that each proof inside a Coq section uses (Section 2). Assuming OPAM has been properly initialized, the following commands install all iCoq components:

```
$ opam repo add pe http://opam-dev.proofengineering.org
$ opam install icoq
```

In particular, this will install an `icoq` executable and include it on the user's PATH. `icoq` is the only file the user needs to be aware of.

3.2 Integration Into a Common Workflow

In theory, iCoq can be integrated into any project using Coq, but the key is to determine where to invoke the `icoq` executable. Here, we consider what we believe is the most common scenario, where a project uses a Makefile to process all `.v` files listed in a file called `_CoqProject` residing in the project repository root directory. The user then simply needs to add the following target to the Makefile:

```
icoq: _CoqProject
        icoq $(shell pwd)
```

The only required argument to `icoq` is the directory that contains the `_CoqProject` file. Other arguments can be added, such as `-timer` to print the execution time summary of various phases, or `-debug` to print debug information. After this integration, regression proof checking is performed by running the command:

```
$ make icoq
```

in the project root directory. The effect of the command is determined by metadata stored in the `.icoq` directory, which is updated after each invocation. When using CISs such as Travis CI [18], this metadata must be persisted across project builds, e.g., via caching.

4 EVALUATION

To evaluate iCoq, we followed the approach traditionally used for evaluating regression test selection techniques [12, 14, 17]. For each project used in the evaluation, we performed the following steps:

(1) Clone the project from its repository (typically on GitHub)
(2) Check out the oldest revision supported by iCoq
(3) Integrate iCoq into the project
(4) While the current revision is supported, do the following:
   (a) Analyze the project to detect affected proofs (all proofs are checked in the first revision)
   (b) Check all affected proofs
   (c) Collect new dependencies
   (d) Check out the subsequent revision in the project history

For each project revision, we stored the log of the execution that contains the number of checked proofs and proof checking time. We extracted these numbers from logs in a separate post-processing phase. Figure 3 shows the proof checking time over 24 revisions (from older to newer) for one of the used projects; we discuss the results in more detail in Section 4.2. We also followed the aforementioned steps (without the iCoq integration, but using the same project revisions) to obtain the number of executed proofs and proof checking time for the traditional Coq toolchain.

We performed the aforementioned steps in two *environments*: CI-Env and LO-Env. The former describes a setting where iCoq is integrated into a project that uses a CIS. This means that timestamps are not preserved across runs, i.e., the traditional toolchain would always check all the proofs in each revision. We persist iCoq dependency metadata across runs in CI-Env; many CISs already support caching directly, and a separate version control repository can also be used in practice. The latter environment (LO-Env) describes a setting where iCoq is used on a local machine. In this setting, the traditional toolchain would use the timestamps and only re-check the changed files (and those files that depend on the changed files). iCoq checks the same proofs in both CI-Env and LO-Env, but in the latter case it avoids quick-compiling unaffected `.v` files from scratch (which saves additional machine time).

4.1 Coq Projects Under Study

We evaluated iCoq on 7 publicly available Coq projects; all but one (Flocq) are on GitHub. We used 4 micro-benchmarks and 3 large verification projects. We used the following criteria when choosing the projects:

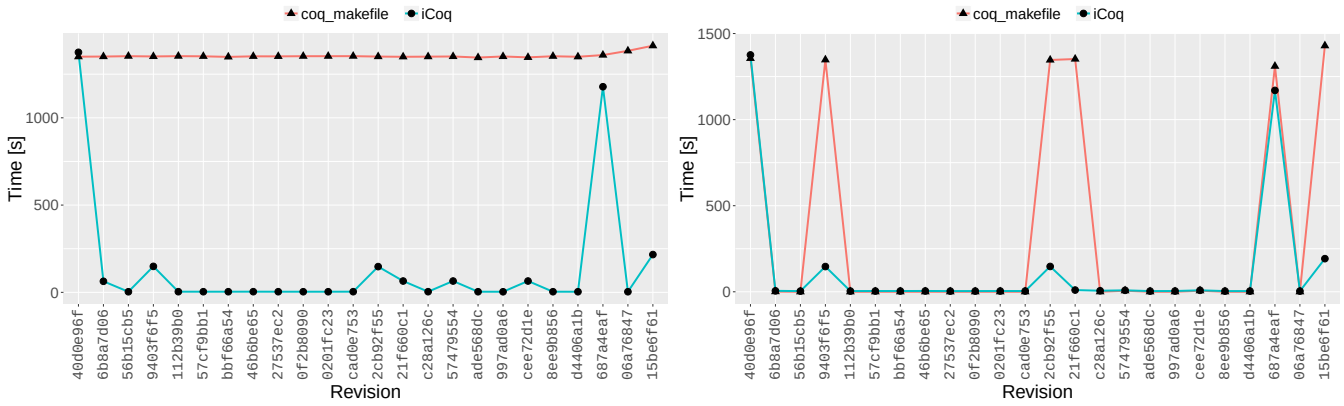
(a) the project should be publicly available;
(b) there should be a range of revisions that can be successfully built with the required software (Section 3);
(c) the project should be non-trivial (and should be popular, e.g., in terms of the number of stars on GitHub); and
(d) our familiarity with the project.

The last criterion was important to simplify the setup and to enable us to confirm the correctness of the tool. To the best of our knowledge, this was the first evaluation of regression proving, and thus we wanted to start with a set of familiar projects. We provide details on the large projects; the micro-benchmarks are described in previous work [9].

Verdi is a framework for verification of distributed systems [20]. We consider revisions from Mar to Jun 2016, which include a verified implementation of the Raft consensus protocol [21]. Each revision comprises over 50k LOC, making Verdi one of the largest publicly available software verification projects.

UniMath is a comprehensive library of formalized mathematics based on the *univalent* interpretation, suggested by Voevodsky, of the types in Coq as so-called homotopy types rather than mathematical sets [19]. The revisions of UniMath under study are from Jan to Mar 2016, and each consist of more than 43k LOC.

Flocq is a Coq library that formalizes floating-point arithmetic in several representations [8]. Flocq is used in the CompCert verified C compiler to reason about programs which use floating-point operations [7]. We considered revisions of Flocq from Jan to Mar 2016, each consisting of more than 22k library LOC.



(a) CI-Env (b) LO-Env

Figure 3: Proof checking time for Verdi over 24 revisions. The plots show proof checking time with the traditional toolchain (`coq_makefile`) and ICoq. We show the results for two modes: CI-Env (left) and LO-Env (right).

4.2 Summary of Results

We run all experiments on a 4-core Intel Core i7-6700 CPU @ 3.40GHz with 16GB of RAM, running Ubuntu 14.04 LTS.

In the CI-Env environment, ICoq speeds up the proof checking time (on average) by 9.62×, 3.44×, and 2.92× for Verdi, UniMath, and Flocq, respectively. Figure 3(a) shows the proof checking time for Verdi in the CI-Env environment over the used revisions.

In the LO-Env environment, ICoq reduced the proof checking time (on average) by 2.60×, 2.23×, and 1.13× for Verdi, UniMath, and Flocq, respectively. Figure 3(b) shows the proof checking time for Verdi in the LO-Env environment over the used revisions.

ICoq reduced the number of checked proofs 21.58×, 4.80×, and 9.62× for Verdi, UniMath, and Flocq, respectively.

4.3 Limitations and Future Work

ICoq inherits the limitations of Coq's asynchronous proof checking toolchain, such as ignoring global universe constraints that pertain to the new mechanism for *generic* definitions in Coq [3]. Consequently, Coq projects that make heavy use of generic definitions are not good targets for the tool. In addition, ICoq currently does not perform dependency analysis of custom tactics defined in the Ltac language that occur in `.v` files. This is not a fundamental limitation, and we expect to address it in future releases of ICoq.

5 CONCLUSION

We described ICoq, the first tool for regression proof selection. ICoq tracks fine-grained dependencies between Coq definitions, propositions, and proofs, and only checks those proofs affected by changes between two revisions. Our evaluation shows that ICoq is up to 3 times faster than Coq's default incremental timestamp-based proof checking toolchain (`coq_makefile`) and up to 10 times faster than checking all proofs from scratch. ICoq can be easily integrated into an existing Coq verification project and used both locally and in continuous integration systems to ensure all ostensibly proven properties actually hold in every project revision.

Acknowledgements. We thank the anonymous reviewers for their comments. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1438982, CCF-1566363, and CCF-1652517.

REFERENCES

[1] 2018. coq-dpdgraph. (2018). https://github.com/Karmaki/coq-dpdgraph.
[2] 2018. Coq manual ch. 15. (2018). https://coq.inria.fr/refman/tools.html.
[3] 2018. Coq manual ch. 28. (2018). https://coq.inria.fr/refman/async-proofs.html.
[4] 2018. The Coq Proof Assistant. (2018). https://coq.inria.fr.
[5] Andrew W. Appel. 2015. Verification of a Cryptographic Primitive: SHA-256. *Trans. Program. Lang. Syst.* 37, 2 (2015), 7:1–7:31.
[6] Bruno Barras, Carst Tankink, and Enrico Tassi. 2015. Asynchronous Processing of Coq Documents: From the Kernel up to the User Interface. In *ITP*. 51–66.
[7] S. Boldo, J. H. Jourdan, X. Leroy, and G. Melquiond. 2013. A Formally-Verified C Compiler Supporting Floating-Point Arithmetic. In *ARITH*. 107–115.
[8] Sylvie Boldo and Guillaume Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *ARITH*. 243–252.
[9] Ahmet Celik, Karl Palmskog, and Milos Gligoric. 2017. iCoq: Regression Proof Selection for Large-Scale Verification Projects. In *ASE*. 171–182.
[10] Haogang Chen, Tej Chajed, Alex Konradi, Stephanie Wang, Atalay İleri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. 2017. Verifying a High-performance Crash-safe File System Using a Tree Specification. In *SOSP*. 270–286.
[11] Michel Chilowicz, Étienne Duris, and Gilles Roussel. 2009. Syntax tree fingerprinting for source code similarity detection. In *ICPC*. 243–247.
[12] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. 2015. Practical Regression Test Selection with Dynamic File Dependencies. In *ISSTA*. 211–222.
[13] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. 2016. Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects. In *ASE*. 426–437.
[14] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. 2016. An Extensive Study of Static Regression Test Selection in Modern Software Evolution. In *FSE*. 583–594.
[15] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (2009), 107–115.
[16] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. 2004. Scaling Regression Testing to Large Software Systems. In *FSE*. 241–251.
[17] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing Regression Test Selection Techniques. *TSE* 22, 8 (1996), 529–551.
[18] TravisCI 2017. Travis CI. (2017). https://travis-ci.org.
[19] Vladimir Voevodsky. 2015. An experimental library of formalized Mathematics based on the univalent foundations. *MSCS* 25, 5 (2015), 1278–1294.
[20] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas Anderson. 2015. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *PLDI*. 357–368.
[21] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. 2016. Planning for change in a formal verification of the Raft consensus protocol. In *CPP*. 154–165.