The Dissertation Committee for Ahmet Celik
certifies that this is the approved version of the following dissertation:

Proof Engineering for Large-Scale Verification Projects

Committee:

Milos Gligoric, Supervisor

Keshav Pingali

Christopher J. Rossbach

Sarfraz Khurshid

Proof Engineering for Large-Scale Verification Projects

by

Ahmet Celik

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2019

To Angel Alya

Acknowledgments

Firstly, I would like to sincerely thank my adviser and friend Milos Gligoric. I would not be graduating without him so quickly. We spent many sleepless nights together working on deadlines supported by free cookies and coffee. We have enjoyed a few conferences together. I cannot pay back for his endless support.

Next, I would like to thank to many exceptional people: my Ph.D. committee members, including Keshav Pingali, Christopher Rossbach, and Sarfraz Khurshid; my collaborators, including Don Batory, Jongwook Kim, Alex Knaust, Young Chul Lee, Aleksandar Milicevic, Sreepathi Pai, Karl Palmskog, Marko Vasic, Kaiyuan Wang, and Chenguang Zhu; and the current and former officemates Allison Berman, Ahsen Dinc, Nima Dini, Chenguang Liu, Pengyu Nie, Marinela Parovic, Zhiqiang Zang, and Mengshi Zhang.

I would like to thank Darko Marinov from the University of Illinois at Urbana-Champaign and his students Alex Gyori, Farah Hariri, Owolabi Legunsen, and August Shi for valuable feedback on several of my (draft) papers. Moreover, I am grateful for Darko's help during the ESEC/FSE 2017 conference in Paderborn, Germany.

I would have not done Ph.D. if Thomas Dillig had not invited me to The University of Texas at Austin. I am thankful to Işıl Dillig and Thomas

Dillig for helping me discover a research area in which I excel and introducing me to Milos.

Lastly, I owe a lot to my family: mom Ayten, dad Durali, and little sister Kübra.

Some chapters in this dissertation include our work that has been published at top Software Engineering conferences. Chapter 2 is published at ASE 2017 [28] and ICSE Demo 2018 [29], and Chapter 3 is published at ISSTA 2018 [120].

Proof Engineering for Large-Scale Verification Projects

Publication No. _____

Ahmet Celik, Ph.D.
The University of Texas at Austin, 2019

Supervisor: Milos Gligoric

Software controls many aspects of our daily lives, thus, software correctness is of utmost importance. One way to develop correct-by-construction software is by using proof assistants, i.e., writing machine-checked proofs of correctness at the level of executable code. Although the obtained guarantees via such development are highly desirable, proof assistants are not currently well adapted to large-scale software development, and are expensive to use in terms of both time and expertise. In particular, the productivity of proof engineers is lowered by inadequate interfaces, processes, and tool support, which lone expert users may not be hindered by, but become serious problems in large-scale projects with many contributors.

This dissertation shows that research in traditional software engineering can improve productivity considerably in large-scale verification projects that use proof assistants and facilitate proliferation of formally verified software.

Specifically, this dissertation, inspired by research in the software engineering area on regression testing, software evolution, and mutation testing, presents three main bodies of research with the goal to speed up proof checking and help proof engineers to evaluate the quality of their verification projects.

First, this dissertation introduces regression proof selection, a technique that tracks fine-grained dependencies between Coq definitions, propositions, and proofs, and only checks those proofs affected by changes between two revisions. We instantiate the technique in a tool dubbed ICoq. 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 proof 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.

Second, this dissertation describes the design and implementation of PICoq, a set of techniques that blend the power of parallel proof checking and proof selection. PICoq can track dependencies between files, definitions, and lemmas and perform parallel checking of only those files or proofs affected by changes between two project revisions. We applied PICoq to perform regression proving over many revisions of several large open source projects. Our results indicate that proof-level parallelism and proof selection is consistently much faster than both sequential checking from scratch and sequential checking with proof selection. In particular, 4-way parallelization is up to 28.6 times faster than the former, and up to 2.8 times faster than the latter.

Third, this dissertation introduces mutation proving, a technique for analyzing quality of verification projects that use proof assistants. We implemented our technique for the Coq proof assistant in a tool dubbed MCoq. MCoq applies a set of mutation operators to Coq functions and datatypes, and then checks proofs of lemmas affected by operator application. We applied MCoq to several medium and large scale Coq projects, and recorded whether proofs passed or failed when applying different mutation operators. We then qualitatively analyzed the failed proofs, and found several examples of weak and incomplete specifications.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software impacts every aspect of our daily lives, thus, software correctness is of utmost importance. Software testing [2] is the most common approach in industry to check software correctness. However, software testing provides very limited correctness guarantees. One way of developing software that is correct-by-construction is by using *proof assistants*, i.e., writing machine-checked proofs of correctness at the level of executable code.

Software developed using proof assistants is reaching unprecedented scale. For example, CompCert C compiler [103], developed and proved correct inside the Coq proof assistant [14, 51], required 8 person-years of effort and 120k lines of code (LOC) to reach its current state, with support for many modern platforms and certified compilation to performant machine code [101]. Similarly, the initial version of the seL4 operating system kernel, proved correct inside the Isabelle/HOL proof assistant [111], required more than 22 person-years of effort and more than 200k LOC [95]. More recent projects based on proof assistants with ambitions to reach the scale of production-level software target critical domains such as file systems [1, 31], security [15, 153], and distributed systems [105, 128, 151]. These projects are now starting to impact

the mainstream software ecosystem: CompCert has found applications in embedded systems [93], and the BoringSSL library, used in the widely deployed Google Chrome Web browser, now includes high-performance cryptographic code verified in Coq [55].

The workflow for software development in proof assistants fundamentally differs from workflows using mainstream programming languages. On one hand, software design and development is driven by the requirements for formal specification and manageable burdens of proof, resulting in unorthodox processes and methodologies [3, 151] and complex organization of components [66, 76]. On the other hand, certain kinds of quality assurance measures, such as unit testing, are simply not applicable [36], while others, such as integration testing, become even more important [62]. Moreover, despite over 40 years of development, proof assistants are not currently well adapted to large-scale software development [17], and are expensive to use in terms of both time and expertise. In particular, *the productivity of proof engineers is lowered by inadequate interfaces, processes, and tool support, which lone expert users may not be hindered by, but become serious problems in large-scale projects with many contributors* [94, 149]. We briefly describe two challenges faced by proof engineers. First, large verification projects may suffer from very long proof checking, which may take days to complete, and this cost is exacerbated for projects that frequently evolve, because proofs need to be rechecked after every change. Second, proof engineers lack a good metric (and a tool) for evaluating the quality of large verification projects.

We believe that research in traditional software engineering can improve productivity considerably in large-scale verification projects that use proof assistants and facilitate proliferation of formally verified software.

This dissertation describes our research effort that leverages software engineering techniques, including *regression testing* [16,127,154], *software evolution*, and *mutation testing* [49,83,88,121], to improve productivity in evolving software verification projects that use proof assistants. More precisely, this dissertation presents three main bodies of research with the goal to *speed up proof checking*, as well as help proof engineers to *evaluate the quality* of their verification projects.

This dissertation make the following key contributions:

⋆ We designed and developed the first technique for large-scale *regression proof selection*, suitable for use in continuous integration services, e.g., Travis CI. We instantiated the technique in a tool dubbed ICoq. ICoq tracks fine-grained dependencies between Coq definitions, propositions, and proofs, and only checks those proofs affected by changes between two revisions. ICoq additionally saves time by ignoring changes with no impact on semantics. 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 showed that proof checking with ICoq is up to 10 times faster than the former and up to 3 times faster than the latter.

- ⋆ We present the first *set of techniques that blend the power of parallel proof checking and proof selection* to speed up regression proving in verification projects, suitable for use both on users' own machines and in workflows involving continuous integration services. We implemented the techniques in a tool, dubbed PICOQ, which supports Coq projects. PICOQ can track dependencies between files, definitions, and lemmas and perform parallel checking of only those files or proofs affected by changes between two project revisions. We applied PICOQ to perform regression proving over many revisions of several large open source projects and measured the proof checking time. While gains from using proof-level parallelism and file selection can be considerable, our results indicate that blended proof-level parallelism and proof selection is consistently much faster than both sequential checking from scratch and sequential checking with proof selection. In particular, 4-way parallelization is up to 28.6 times faster than the former, and up to 2.8 times faster than the latter.

- ⋆ We designed and implemented the first *mutation proving* technique for evaluating and enhancing the quality of verification projects that use proof assistants. We define a set of mutation operators on functions and data, inspired by mutation operators defined previously for functional and imperative programming languages. We implemented mutation proving in a tool, dubbed MCOQ, which supports Coq projects. Our tool brings significant extensions to Coq and the SERAPI library for serialization and deserialization of Coq syntax [63]; these extensions pave the way for other transformations of Coq

code. To make mutation proving of large projects feasible in practice, we optimized MCoq in several ways to make it run faster. In particular, we implemented several novel forms of parallel checking of affected proofs. We performed an empirical study using MCoq on 12 large and medium-sized open source Coq projects. For each project, we recorded the number of generated and killed mutants and the execution time. We qualitatively analyzed a subset of the live mutants and found several instances of weak and incomplete specifications. Our work resulted in many improvements and bug fixes to SerAPI and enhanced its robustness when applied to large-scale Coq projects, showing that complex, extensible proof documents can be manipulated in a lightweight way. We made several modifications to Coq itself, and these changes have been accepted by Coq developers.

Chapter 2

iCoq: Regression Proof Selection for Large-Scale Verification Projects[1]

Proof assistants such as Coq are used to construct and check formal proofs in many large-scale verification projects. As proofs grow in number and size, the need for tool support to quickly find failing proofs after revising a project increases. We present a technique for large-scale regression proof selection, suitable for use in continuous integration services, e.g., Travis CI. We instantiate the technique in a tool dubbed iCoq. iCoq tracks fine-grained dependencies between Coq definitions, propositions, and proofs, and only checks those proofs affected by changes between two revisions. iCoq additionally saves time by ignoring changes with no impact on semantics. 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.

[1]Parts of this chapter are published at ASE 2017 [28] and ICSE Demo 2018 [29]. My contributions span all aspects of this work: defining the project, designing a solution, implementing a tool, performing evaluation, and documenting findings.

2.1 Overview

Verification projects based on construction and certification of formal proofs inside proof assistants have reached a hitherto unprecedented scale. Large projects take two main forms: formalizations of mathematical theories and programs with accompanying proofs of correctness at the level of executable code [67]. The former includes the proofs of the four-color theorem [73] and the Feit-Thompson odd order theorem in Coq [74], and a proof of the Kepler conjecture in HOL Light [82]; the latter includes the certified seL4 operating system kernel in Isabelle/HOL [95], and the CompCert C compiler in Coq [103].

Using proof assistants has advantages with respect to scalability, modularity, and reliability compared to using more automated methods based only on model checking or SMT solving [67]. On the other hand, proof assistants are more human resource intensive to use than model checkers, and come with less tool support than what is available to programmers using mainstream programming languages. Specifically, Wenzel has recently noted the need for more systematic tool support to maintain repositories of formal proofs [149].

Large verification projects based on proof assistants are similar to regular software projects in that (a) the end goal is a software artifact with certain properties, (b) developers use an integrated development environment (IDE) to write code, which is then checked by a tool and submitted to a version control system shared with others. Evidence from earlier undertakings indicate that such projects require engineering effort similar to, or beyond, some of the

most complex software projects; for example, the proof of the odd order theorem in Coq was a six-year effort of a team of 15 people, resulting in 170,000 lines of code [114].

We believe that proper tool support for large-scale *proof engineering* using proof assistants is an important and growing concern [94]. In particular, it is important to quickly find and report errors in *evolving* Coq and Isabelle/HOL projects. However, just as for large projects in, e.g., Java, determining the errors caused by a particular change can be a time-consuming process. For instance, the Coq correctness proofs of an implementation of the Raft distributed consensus protocol [115] are around 50k lines in total [151] and take more than 30 minutes to check from scratch on a computer with an Intel Core i7 4th generation processor. Potentially, a Coq user has to wait all this time to find out whether a change in some definition makes a seemingly unrelated proof fail.

Until recently, all proof assistants in the LCF family, including Isabelle/HOL and Coq, relied on user interaction through a read-eval-print loop inherited from their predecessor. This interaction model effectively prevents event-based user interaction with proof assistant files inside an IDE, in the style of Eclipse. Initial work in Isabelle/HOL to address this problem [148] paved the way for recent architectural changes in Coq towards a *document-oriented* interaction model, where the proof assistant backend asynchronously receives definitions, proof commands, and proof checking tasks from the user, all of which may concern disparate parts of a project [9].

In this chapter, we show that potential gains in productivity from Coq's new interaction model go beyond recent application inside IDEs [58]. We present iCoq, a tool for *regression proof selection* for large-scale Coq projects, suitable for use in workflows involving version control and continuous integration services (CISs), e.g., Travis CI [85, 137]. (CISs run tests/proofs of a project whenever code of the project changes. These services have become widely used; Travis CI, one out of more than 20 available CISs, is used by more than 300k projects [87].) iCoq works by tracking dependencies between 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. In addition, iCoq saves time by ignoring changes with no impact on the semantics of files, e.g., additions of comments or whitespaces.

Our approach is based on a fundamental analogy between *tests* and *proofs*. As Beck has noted in context of extreme programming [11], a test can be viewed as a method that checks a *partial functional specification* of a system. Consequently, a proposition about a (pure) function in Coq's logic, along with its proof, can be viewed as an amalgamation of many—possibly an infinite number of—tests. For example, changing the definition of a function in a Coq file can potentially impact many proofs, analogously to how changes in Java programs affect tests in a test suite. Using this analogy, iCoq mirrors previous work in *regression testing* for mainstream programming languages, in

particular techniques for lightweight *regression test selection*, which have been shown to significantly lower the cost of running test suites, and hence find errors more quickly [16, 54, 69, 100, 117, 118, 125, 127, 136, 154]. Such tools have recently been adopted by many large open-source Java projects. ICoq opens the door for similar benefits to accrue to developers of large Coq projects.

Nevertheless, proofs and tests are also different in several important ways. First, the proof of one claim typically depends on other claims; tests are typically completely independent of other tests. Second, function definitions, claims, and proof scripts are often interspersed in Coq files; test code is seldom interspersed with program code. Third, Coq proof checking is done in the *same environment* as the processing of definitions and even computation; executing tests is usually done completely separately from code compilation. We overcome these three challenges by leveraging Coq's newly-added toolchain for asynchronous proof processing [9].

To evaluate ICoq, we applied it on revision histories of several Coq developments, including three large-scale projects, and measured the time savings compared to proof checking from scratch (typical use in continuous integration systems) and incremental proof checking using Coq's timestamp-based toolchain (typical command-line use). Our results show that processing proofs with ICoq is up to 10× faster than the former, and up to 3× faster than the latter.

This chapter describes the following contributions:

⋆ **Technique**: We propose regression proof selection (inspired by regression test selection), a technique that can substantially reduce proof checking time for evolving verification projects. To the best of our knowledge, this is the first application of research in regression testing to the domain of formal proofs. Our insight is that due to simpler language features in proof assistants than in imperative languages (e.g., Java), regression proof selection can straightforwardly collect fine-grained dependencies, which are used to identify proofs to recheck at each project revision.

⋆ **Tool**: We implemented regression proof selection in a tool, dubbed iCoq, which supports Coq projects. We provide a version of our tool on the following URL: `http://cozy.ece.utexas.edu/icoq`.

⋆ **Evaluation**: We performed an empirical study to measure the effectiveness (in terms of both number of executed proofs and proof checking time) of regression proof selection using iCoq. We used several open-source Coq projects, including three large-scale projects.

2.2 Coq Background

The Coq proof assistant can be viewed as, on the one hand, a small and powerful purely functional programming language, and on the other hand, a system for specifying properties about programs and proving them. Coq is based on a constructive type theory called the Calculus of Inductive Constructions (CIC) [123]. In CIC, both programs and propositions about programs

are types inhabited by *terms*, in effect putting program construction and proving on the same footing. Via a frontend, e.g., emacs with Proof General [6], a user interactively constructs tentative proof terms for propositions (assertions) using operations called *tactics*, and the final result is only accepted after Coq's type checker was run successfully by the backend on the term. Barring use of inconsistent axioms and frontend issues, the user need only trust that the comparatively small type checking kernel is correctly implemented and compiled to trust the results. The interactive proof development process in Coq is illustrated in Figure 2.1.

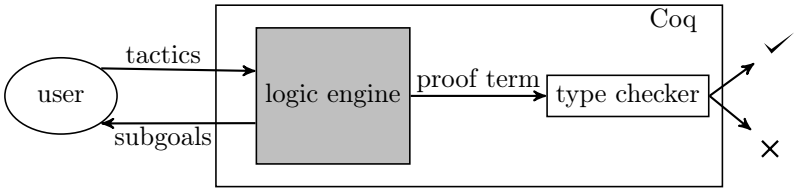


Figure 2.1: Coq interactive proof development overview.

Definitions of functions and lemmas processed by Coq are written in the Gallina language, and reside in files ending in `.v`. The standard Coq batch proof processing ("compilation") tool, `coqc`, takes a `.v` file as input and produces a `.vo` file as output that contains full binary representations of processed Gallina constructs, including proofs. If the proofs are large and complex, `.vo` files can be tens of megabytes large [107]. Since files may depend on other files, checking all proofs in a Coq project requires some form of dependency analysis. The standard `coq_makefile` tool generates a Makefile which, by default, calls the `coqdep` tool for this purpose [44]. `coqdep` builds

a dependency graph for all input files based on simple syntactic analysis of `Require` commands (similar to `import` statements in Java) in files, which indicate direct dependency at the file level. When proof checking is then performed via the Makefile, the generated dependency graph is used to compile `.v` files in some allowed order, possibly in parallel. The generated Makefile also enables timestamp-based incremental processing of Coq projects, which is known to be limited [56, 70].

Figure 2.2, Figure 2.3, and Figure 2.4 show the content of three example Gallina files, where a simple function on lists of natural numbers is defined, specified, and proved correct. `Alternate.v` contains definitions used in the two other files, and these dependencies are found by `coqdep`. The dependency data is used to generate a Makefile that calls `coqc` to produce `.vo` files; if `Alternate.v` is subsequently modified in any way after compilation, the other files will also be automatically recompiled when running `make`. On the other hand, modification of the other files does not trigger recompilation of `Alternate.v`.

In effect, the `coqdep` tool produces a coarse-grained dependency graph of a Coq development at the level of `.v` files, as shown in Figure 2.5(a) for the example Gallina files; dashed arrows indicate dependencies on files from Coq's standard library, which are usually disregarded. Internally, Coq maintains a fine-grained dependency graph at the level of constants, reminiscent of the graph shown in Figure 2.5(b).

In each Coq file, the commands between `Proof`. and `Qed`. are *proof*

```
Require Export List. Export ListNotations.

Fixpoint alternate l1 l2 : list nat :=
match l1 with
| [] ⇒ l2 | h1 :: t1 ⇒
 match l2 with
 | [] ⇒ h1 :: t1 | h2 :: t2 ⇒ h1 :: h2 :: alternate t1 t2
 end
end.

Inductive alt : list nat → list nat → list nat → Prop :=
| alt_nil : ∀ l, alt [] l l
| alt_step : ∀ a l t1 t2,
   alt l t1 t2 → alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  ∀ l1 l2 l3, alt l1 l2 l3 → alternate l1 l2 = l3.
Proof.
induction l1; intros.
- inversion H. subst. simpl. reflexivity.
- destruct l2; simpl; inversion H; inversion H4; auto.
  apply IHl1 in H9. rewrite H9. reflexivity.
Qed.
```

Figure 2.2: Coq Gallina a file example `Alternate.v`.

scripts comprised of tactic calls along with bullets to indicate goal structure. Proof scripts instruct Coq how to build a proof term. Tactics can be pipelined and may perform sophisticated and time-consuming search operations, splitting of goals, and term rewriting. Ultimately, tactics produce a proof t in Coq's term syntax, of which a fragment is shown in Figure 2.6. For example, the beginning of the proof of `alt_alternate` can be represented as

$$\mathsf{Const}(\mathsf{Lambda}(\texttt{l1}, \mathsf{App}(\texttt{list}, \texttt{nat}), \mathsf{App}(\texttt{list\_ind}, \ldots)))$$

where `list` and `nat` are the Ind terms for the algebraic datatypes for polymorphic lists and natural numbers, respectively, and `list_ind` is the Const term

```
Require Import Alternate.

Lemma alt_∃ : ∀ l1 l2, ∃ l3, alt l1 l2 l3.
Proof.
induction l1; intros; destruct l2.
- ∃ []. apply alt_nil.
- ∃ (n :: l2). apply alt_nil.
- ∃ (a :: l1). apply alt_step. apply alt_nil.
- specialize(IHl1 l2). destruct IHl1. ∃ (a :: n :: x).
  repeat apply alt_step. auto.
Qed.
```

Figure 2.3: Coq Gallina a file example `AltLem.v`.

for a list induction principle.

Coq version 8.5, the first stable release to include architectural changes to support a document-oriented interaction model [9], 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`. Only the type (assertion) of an opaque identifier such as `alt_alternate`, i.e., not the body term, can be referenced in other parts of a Coq development, whence type checking of all such terms can normally be performed in complete isolation. Specifically, `.vio` files contain *proof-checking tasks*, which can be performed individually 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 in most developments, but must then assume that all proofs are correct.

For example, the lemma `alternate_correct` (`AlternateLem.v`) in the Coq development in Figure 2.4 depends only on the types (assertions)

```
Require Import Alternate.

Lemma alternate_alt :
  ∀ l1 l2 l3, alternate l1 l2 = l3 → alt l1 l2 l3.
Proof.
induction l1; simpl; intros.
- rewrite H. apply alt_nil.
- destruct l2; subst; apply alt_step; try apply alt_nil.
  apply alt_step. apply IHl1. reflexivity.
Qed.

Lemma alternate_correct :
  ∀ l1 l2 l3, alternate l1 l2 = l3 <→ alt l1 l2 l3.
Proof.
intros; split; [apply alternate_alt | apply alt_alternate].
Qed.
```

Figure 2.4: Coq Gallina a file example `AlternateLem.v`.

of `alternate_alt` and `alt_alternate`, but not their *proofs*; consequently, the proof of `alternate_correct` need not be re-checked if only the proof of `alt_alternate` is changed. In this case, the sole required action is to re-check the proof of `alt_alternate`, which can be accomplished by first quick-compiling `Alternate.v` and then running the single proof-checking task in `Alternate.vio`. Figure 2.7 illustrates the possible workflows for `Alternate.v` made possible by Coq's document-oriented model.

Coq uses a notion of *sections* to organize common assumptions made in a collection of lemmas, say, that equality on type `A` is decidable (`A_eq_dec`). A lemma may reference one or more such assumptions, which then become quantified variables that must be instantiated when the lemma is referenced outside of the section. However, by default, Coq only determines the used section variables of a lemma when the end of the section is reached. This

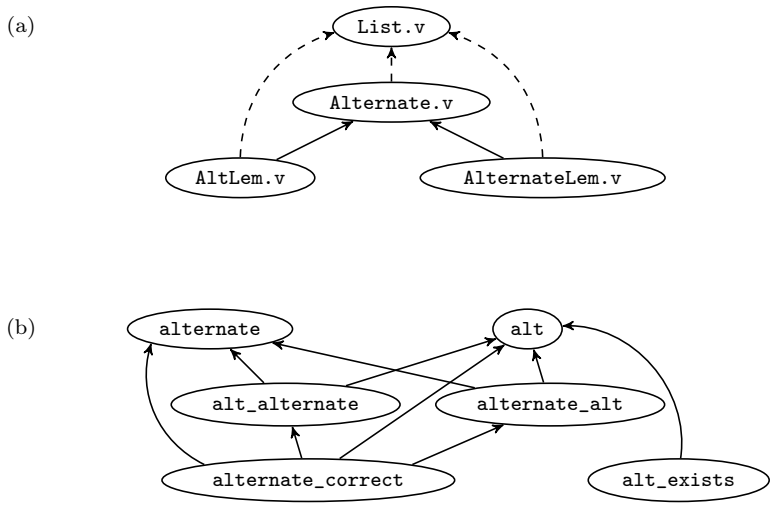


Figure 2.5: Coarse- vs. fine-grained dependency graphs for example Coq development.

$$t \quad ::= \mathsf{Var}(x) \mid \mathsf{Prod}(n, t, t') \mid \mathsf{Lambda}(n, t, c) \mid \mathsf{App}(c, ca) \mid$$
$$\mathsf{Const}(c) \mid \mathsf{Ind}(i) \mid \mathsf{Construct}(cs) \mid \mathsf{Fix}(f) \mid \ldots$$

Figure 2.6: Coq term syntax fragment.

means that the final type (assertion) of the section lemma is not known when considered in isolation, whence its proof cannot be immediately checked as an asynchronous task. To get around this problem, Coq allows section lemmas to be annotated with the assumptions they use (e.g., `Proof using A_eq_dec`). The required annotations can be derived from metadata produced by Coq during compilation of source files to `.vo` files [133], and then inserted back into the source files. In the evaluation of our technique, we used this approach to add annotations to all revisions of the projects under study as a separate initial step.

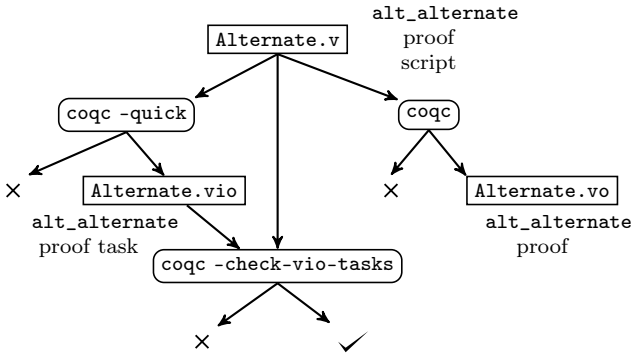


Figure 2.7: Coq workflows for `Alternate.v`.

2.3 Technique

This section describes our proof selection technique. We first describe its phases at a high level, then details on the lower-level steps, and finally our implementation in the ICoq tool. The key idea is to incrementally build and analyze both coarse-grained and fine-grained dependency graphs to produce the minimal set of proofs that need to be checked after a change has been made to a project. The advantage of our technique compared to the timestamp-based incremental processing of files stems from that, generally, checking a few proofs in isolation spread out across a development takes much less time and effort than checking all proofs in all affected source files.

2.3.1 Phases

Roughly, our technique follows the three phases of a typical regression test selection technique [154]: an initial *analysis* phase that locates proofs affected by recent changes, followed by an *execution* phase that checks the selected proofs, followed by a final *collection* phase which produces dependen-

cies for the next revision. We assume that both the file-level and lemma-level dependencies and checksums of the last revision of the project are available at the start of the initial phase.

Analysis phase: First, for each source file in the project, we check whether its checksum is still the same since the last revision. Then, we perform file-level dependency analysis and build an up-to-date coarse-grained dependency graph that includes checksums, with changed files marked. This graph is then used to quick-compile the changed source files, allowing us to compute checksums of the term representations of individual definitions and lemma statements that may have changed. At the same time, we also determine the proof tasks available in each changed source file, and compute the checksum of each proof script associated with a proof task. Using our knowledge of proof tasks and checksums for fine-grained entities, we obtain a fine-grained dependency graph where each modified entity is marked, and from which recently removed entities are purged.

By going through all modified entities in the fine-grained dependency graph, we then calculate the transitively impacted entities, and mark them in the graph. The set of proof tasks to execute is then precisely the tasks associated with the set of modified and impacted entities. Note that this process of discovering impacted proofs is similar to the process of "invalidating the upward transitive closure" in some build systems, e.g., Bazel [10].

Execution phase: Given the list of proof tasks and their associated source files and binary quick-compiled files from the previous phase, we emit the

commands for checking those tasks. After each command is executed, we note the dependencies of the proof on other lemmas and definitions; this information is only available when the proof term has actually been constructed and stored in memory.

Collection phase: This phase finds the dependencies of all modified definitions and lemmas by extracting them from the quick-compiled files and combining the results with the proof dependencies obtained in the previous phase. We use these dependencies to build a complete up-to-date fine-grained dependency graph that includes checksums. We then store this graph as a file, to be used in the analysis phase of the next project revision.

Running example: We exemplify our technique for Coq using the code in figures 2.2, 2.3, and 2.4. Assume that we integrated ICoq in the project at revision v1. At that revision, we compute the checksums of all `.v` files, run `coqdep` on them, and build the graph shown in Figure 2.5(a); no checksums existed in revisions prior to v1 and therefore the current values are considered different by definition. Since all file checksums are different, we quick-compile all files into `.vio` files and compute all the checksums for all definitions and lemma statement terms. Then, we note the proof tasks in each file and compute checksums for the associated proof scripts. Again, all checksums are different by definition, so we check the proofs of all lemmas (`alt_alternate`, `alt_exists`, `alternate_alt`, and `alternate_correct`). From the corresponding proof terms, and the terms for `alternate` and `alt`, we construct the graph in Figure 2.5(b) and add checksums for all nodes. The graphs and checksums are

```
Fixpoint alternate (l1 l2 : list nat) : list nat :=
match l1, l2 with
| [], _ ⇒ l2 | _, [] ⇒ l1
| h1 :: t1, h2 :: t2 ⇒ h1 :: h2 :: alternate t1 t2
end.
```

Figure 2.8: Modified Coq function definition in `Alternate.v`.

then stored for future use.

Suppose that the developer of the example Coq project rewrites the definition of the function `alternate` to the one in Figure 2.8; this change leads to a new revision v2 of the project. At the file level, the checksum of `Alternate.v` becomes different from before. However, `coqdep` reveals that the file dependency graph is still the same as in Figure 2.5(a). Since the other `.v` files depend on `Alternate.v`, we compile all `.v` files into `.vio` files in some order allowed by the graph. After then computing checksums of terms (using `Alternate.vio`) and proof scripts (using `Alternate.v`), we conclude that only (the body of) `alternate` has been modified. Using this information and the graph in Figure 2.5(b), we determine that the proofs of `alt_alternate`, `alternate_alt`, and `alternate_correct` are impacted and must be checked. Consequently, we run the commands to check these proofs (while `alt_exists` is not checked, because it was not impacted).

After each proof checking task has completed, we note that no dependencies in the proofs have changed. Finally, we extract and analyze dependencies from the only modified non-proof term (`alternate`), confirming that the graph in Figure 2.5(b) is up-to-date after the new checksum for `alternate`

has been added.

2.3.2 iCoq Components and Workflow

Our current implementation of the technique is written in OCaml, Java, and Bash. We developed a number of separate Coq tools and plugins. Since Coq developments are not upwards or downwards compatible in general, we target Coq version 8.5 to support the largest range of project revision histories susceptible to asynchronous proof checking; we expect no fundamental issues with supporting future Coq versions. Our tools and plugins can also be used (and be useful) outside the context of iCoq, as demonstrated in later chapters of this dissertation.

`coq-depends` **plugin**: To extract dependencies from compiled Coq files (`.vo` and `.vio`), we adapted and extended previous work on the `coq-dpdgraph` Coq plugin [40], which builds dependency graphs for given identifiers or modules (files). In essence, the derived plugin, called `coq-depends`, traverses a Coq term abstract syntax tree (AST), and records the globally unique ("kernel") name of all referenced identifiers it encounters, such as those of inductive types, lemmas, and functions. By performing the dependency extraction at the level of ASTs in the Coq backend, our tool is isolated from complexities at the Gallina level, such as custom notations and implicit arguments. In contrast to `coq-dpdgraph`, `coq-depends` does not perform recursive dependency extraction, and supports `.vio` files, which do not contain the proofs of opaque identifiers that `coq-dpdgraph` expects to be present. The plugin makes no

distinction between depending on an identifier of a lemma or function that is inside the scope of a project or outside it. In particular, if there is a dependency on a lemma in the Coq standard library, which is normally assumed to be stable across revisions, it must be filtered out from the plugin output to be excluded from analysis. For example, from the proof term for the lemma `alt_alternate` described in Section 2.2, `coq-depends` extracts the set of kernel names {`Alternate.alt`, `Alternate.alternate`, `Coq.Init.Datatypes.list`, `Coq.Init.Datatypes.list_ind`, ...}. Here, to filter out unnecessary dependencies, it suffices to exclude names with the prefix "`Coq.`".

`coq-ast` **plugin**: To compare Coq identifiers across project revisions, we developed a plugin for computing short summaries (digests) of Coq term ASTs that capture the *structure* of the trees. We use a technique for computing summaries based on cryptographic hashes that was shown to be effective at programming language syntax fingerprinting by Chilowicz et al. [34]. More specifically, letting $\mathcal{C}$ be a hashing function, $\cdot$ the string concatenation operation, t a term AST with root node r and child trees $t_1, \ldots, t_n$, and V a function from AST nodes to strings, Chilowicz et al. define a hash function $\mathcal{H}_{\mathcal{C}}$ such that $\mathcal{H}_{\mathcal{C}}(t) = \mathcal{C}(V(r) \cdot \mathcal{H}_{\mathcal{C}}(t_1) \cdot \ldots \cdot \mathcal{H}_{\mathcal{C}}(t_n))$. Note that this function, which we implemented in OCaml with $\mathcal{C} = \text{MD5}$, is asymptotically linear in the number of nodes in the tree.

The function V is defined in an obvious way based on the syntax in Figure 2.6; as an example, Figure 2.9 shows a fragment of the AST of the proof of `alt_alternate` in `Alternate.v` where V has been applied to each node.

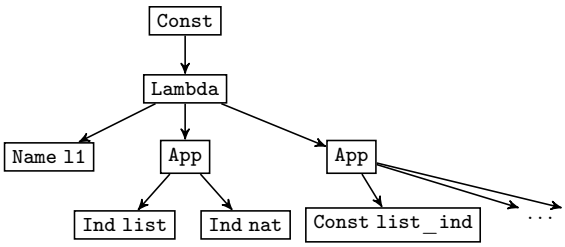


Figure 2.9: AST with string values of nodes for example Coq term.

To keep ASTs as shallow as possible, we do not unfold bodies of referenced inductive types or constants, and simply use their (unique) kernel names.

`coqdigest` **tool**: Since we cannot compute digests of ASTs of opaque identifiers without actually performing all the proof-checking work (that we are trying to skip), we use digests of the actual proof scripts ("tactic soups") in the `.v` files. From the standard `coqdoc` tool which translates `.v` files into documentation, we derived a tool dubbed `coqdigest` that extracts the proof scripts of opaque lemmas while ignoring sequences of characters that do not affect semantics, and returns the MD5 hash of the results. The tool also notes whether a lemma is *admitted*, i.e., whether an identifier with an unfinished proof is assumed as complete for the rest of the development; this is a common device used in early phases of verification projects.

For example, when parsing `AlternateLem.v` from Figure 2.4, `coqdigest` determines that there are two proof tasks in the file, one for the lemma `alternate_alt` and one for the lemma `alternate_correct`. For the latter specifically, `coqdigest` computes the MD5 hash of the proof script

```
intros; split; [apply alternate_alt | apply alt_alternate].
```

`coqc` **dependency extraction extension**: A proof term for a proof task in a `.vio` file is only available when the proof task completes. Yet, to properly update the identifier dependency graph for the next revision, all dependencies must be extracted from such terms. Consequently, we extended the `coqc` tool with an additional command that, when given a `.vio` file, its associated `.v` file, and a proof task, checks the task and then outputs all the dependencies in the proof term using the technique from `coq-depends`. Due to how the proof checking interface works in Coq 8.5, accessing the proof term is only possible when the proof is complete, i.e., has not been admitted. For this reason, ICoq ignores checking proofs of admitted lemmas, although changes in their statement (type) can lead to checking of other proofs that depend on them. Since our `coqc` extension only uses the existing proof checking facilities, it does not affect the soundness of Coq.

Dependency graph builder and analyzer: We implemented our own dependency graph builder and dependency analysis in Java. The resulting program reads files (mostly 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.

Toolchain workflow: If all proofs in a `.v` file need to be checked, compiling a `.vo` file is usually significantly faster than first producing a `.vio` file and then executing all proof tasks. Consequently, we compile all `.v` files in the initial revision of a project into `.vo` files, and via those, extract dependencies directly from both proofs and definitions. For subsequent revisions,

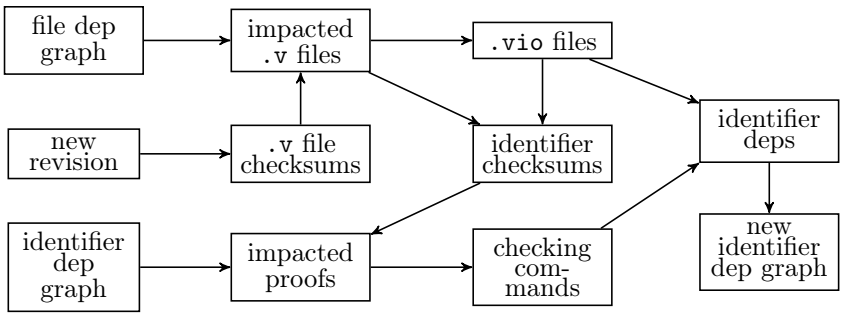


Figure 2.10: Toolchain workflow.

the toolchain workflow (illustrated in Figure 2.10) follows the general steps of the technique outlined in Section 2.3.1. First, the Java program reads the JSON representations of the file-level and identifier-level dependency graphs from the last revision. Then, it computes checksums of all `.v` files in the revision, runs `coqdep` on changed files, parses the output, and updates the file-level dependency graph. Using the graph, the program calls `coqc` to quick-compile all impacted files into `.vio` files. Then, it runs `coqdigest` on all new and changed `.v` files, and `coq-ast` on their `.vio` counterparts, obtaining (via parsing of JSON files) checksums for all identifiers and a list of proof tasks. This is sufficient to enable marking all impacted identifiers in the dependency graph. From the updated graph, the program obtains and runs all proof tasks associated with impacted identifiers using the extended `coqc` command, and then parses and incorporates the JSON output into the fine-grained dependency graph. Finally, it uses `coq-depends` to obtain the dependencies of all impacted non-proof identifiers, writes the up-to-date graph to disk along with the coarse-grained file-level graph.

2.4 Evaluation

To assess the usability of iCoq on large verification projects, we answer the following research questions:

RQ1: How effective is iCoq (compared to the state-of-the-art techniques), i.e., what is the reduction in the number of checked proofs?

RQ2: How effective is iCoq in terms of the proof checking time in a continuous integration environment?

RQ3: How effective is iCoq in terms of the proof checking time outside a continuous integration environment (i.e., for verification on a user's machine)?

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

2.4.1 Verification Projects Under Study

Table 2.1 shows the list of projects used in our study; all projects are publicly available, all but one on GitHub [68]. We selected projects based on (a) public availability of their revision history during principal development, (b) compatibility of their revision history with Coq 8.5, (c) their size and popularity, and (d) our familiarity with their codebases; the latter was necessary for a successful experimental setup. For each project, we list the name, reference the repository location, and show the last revision/SHA we used for our experiments, the number of lines of Coq code (LOC) for the last revision (as reported by `cloc` [38]), and the number of revisions for the experiments.

Table 2.1: Verification Projects Used in the Evaluation.

Project	**URL**	**SHA**	**LOC**	**#Revs**
CTLTCTL	[46]	ac57a84f	601	10
InfSeqExt	[86]	5a52a76f	1756	10
StructTact	[131]	8f1bc10a	2496	10
WeakUpTo	[145]	e570e6dc	1819	10
Flocq	[60]	4161c990	24786	24
UniMath	[139]	5e525f08	43049	24
Verdi	[141]	15be6f61	53939	24
$\sum$	N/A	N/A	128446	112
Avg.	N/A	N/A	18349.42	16

Based on projects' characteristics, we say that the first four projects are *micro-benchmarks*, and the other three projects are large-scale proof developments.

Verdi and Verdi Raft: Verdi is a framework for verification of implementations of distributed systems [150]. While the framework is not currently tied to any one particular verification project, it was initially bundled with a verified implementation of the Raft distributed consensus protocol [151]. We consider revisions from Mar to Jun 2016, before Verdi and the Raft implementation were separated. Each revision comprises over 50k LOC, making Verdi one of the largest publicly available software verification projects. Many Verdi proofs use extensive custom tactic-based automation; the resultant long proof-checking time was one of the initial motivations for developing ICoq.

UniMath: 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 [142]. The

revisions of UniMath under study are from Jan to Mar 2016, and each consist of more than 43k LOC.

Flocq: Flocq is a Coq library that formalizes floating-point arithmetic in several representations [21], e.g., as described in the IEEE-754 standard. Flocq is used in the CompCert verified C compiler to reason about programs which use floating-point operations [20]. We considered revisions of Flocq from Jan to Mar 2016, each consisting of more than 22k library LOC.

2.4.2 Variables

In the following subsections, we document the independent and dependent variables used in our study.

Independent variables: We manipulate two independent variables: *proof checking techniques* and *the development environment*. Regarding the proof checking techniques, we use (a) Coq's timestamp-based toolchain that we described in Section 2.2 (we refer to this technique as `coq_makefile`), and (b) ICoq that implements regression proof selection. Our development environments include CI-Env and LO-Env. CI-Env describes an environment that uses a Continuous Integration Service (CIS) to check proofs. Note that a CIS checks proofs in a clean environment for each revision. LO-Env describes an environment where developers use their local machines to check proofs. Note that file timestamps are preserved in the latter case, but not in the former.

Dependent variables: Our dependent variables measure the effectiveness of proof selection techniques at reducing the amount of effort required to reproof

modified programs. To do this we compute the *proof selection percentage* and measure the *proof checking time*. The proof selection percentage is derived from the ratio of selected proofs to the total number of available proofs executed by `coq_makefile` in the CI-Env environment. We use P^{sel} to denote this variable. Proof checking time is measured as the *end-to-end time* that includes *all* phases (described in detail in Section 2.3) of iCoq.

2.4.3 Experiment Procedure

Figure 2.11 illustrates our experiment procedure that collects the data necessary to answer our research questions. As input, the procedure accepts one of the projects under study (Table 2.1), a number of revisions to be used in the experiment, and a development environment (either CI-Env or LO-Env). In the initial step (line 2), the procedure clones the project repository from the URL in Table 2.1. Next, the procedure iterates over the latest κ revisions, from the oldest to the newest revision. In each iteration of the loop, the procedure (a) obtains a copy of the project for the current revision (line 4), (b) configures the project (as the preparation for the proof checking), and (c) selects proofs that are affected by changes and checks those proofs. Finally, if the procedure is simulating the CI-Env, the timestamps of all files have to be updated.

It is important to observe that we need to save dependency files for iCoq between two revisions. Recently, many CISs have started supporting caching [130, 143], which we can utilize to store the dependencies. Considering that caching is fast and iCoq's dependency files are small, we do not associate

```
Require: p a project under study
Require: κ the number of revisions
Require: ε a development environment
 1: procedure ExperimentProcedure(p, κ, ε)
 2:     Clone(p.url)
 3:     for all ρ ∈ LatestRevisions(κ, p) do
 4:         Checkout(ρ)
 5:         Configure(p)
 6:         SelectExecuteAndCollect(p)
 7:         if ε = CI-Env then
 8:             TouchFiles(p)
 9:         end if
10:     end for
11: end procedure
```

Figure 2.11: Experiment procedure.

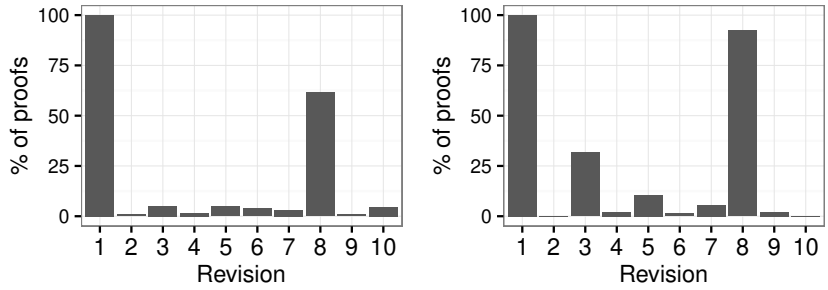
any overhead with keeping dependencies in the CI environment.

One of the key steps in the experiment procedure is to select and check proofs (line 6). During this step, our procedure stores the execution logs, which include the list of selected proofs and the proof checking time. We analyze these logs in the following subsection to answer our research questions.

2.4.4 Results

We obtained all necessary data by invoking the procedure in Figure 2.11 twenty-eight times: one invocation for each project in Table 2.1, two proof checking techniques (`coq_makefile` and iCoq), and two environments (CI-Env and LO-Env). In total, we selected and checked proofs on 112 revisions.

RQ1: Figure 2.12 shows the proof selection percentage for two (out of four) micro-benchmarks. We can observe substantial reduction in the number of executed proofs at many revisions. Overall, across all revisions, we find that



(a) InfSeqExt (b) StructTact

Figure 2.12: Proof selection percentages for two micro-benchmarks.

iCoq executes 226 (on average 22.60) and 398 (on average 39.80) proofs for InfSeqExt and StructTact, respectively. On the other hand, we find that the `coq_makefile` technique executes 1,240 (on average 124.00) and 1,635 (on average 163.50) proofs for InfSeqExt and StructTact, respectively. In other words, iCoq reduces the number of checked proofs by 81.78% and 75.66% for InfSeqExt and StructTact, respectively.

Although we obtained proof selection percentages for the other two micro-benchmarks (WeakUpTo and CTLTCTL), we do not show these numbers because the developers of the projects have not changed any code in the last 10 revisions. As expected, iCoq has not selected any proofs for execution. Note that open-source projects have frequent non-code changes that have no impact on tests/proofs [69]; these changes can include changes in documentation and metadata files.

Finally, we show the results for the three largest projects used in our study. We format the results slightly differently for several reasons, including

Table 2.2: Total and Average Number of Selected Proofs for Large Verification Projects.

Project		**Proofs iCoq**	**Proofs Total**	P^{sel}
Flocq	$\sum$	2164	22482	N/A
	Avg.	90.16	936.75	9.62
UniMath	$\sum$	853	17754	N/A
	Avg.	35.54	739.75	4.85
Verdi	$\sum$	4458	65413	N/A
	Avg.	185.75	2725.54	6.80
	Revision	**iCoq**	**Total**	P^{sel}
Verdi (details)	40d0e96f	2748	2748	100.00
	6b8a7d06	0	2748	0.00
	56b15cb5	0	2748	0.00
	9403f6f5	2	2750	0.07
	112b39b0	0	2750	0.00
	57cf9bb1	0	2750	0.00
	bbf66a54	0	2750	0.00
	46b6be65	0	2750	0.00
	27537ec2	0	2750	0.00
	0f2b8090	0	2750	0.00
	0201fc23	0	2750	0.00
	cad0e753	0	2750	0.00
	2cb92f55	2	2750	0.07
	21f660c1	3	2697	0.11
	c28a126c	0	2697	0.00
	57479554	3	2697	0.11
	ade568dc	0	2697	0.00
	997ad0a6	0	2697	0.00
	cee72d1e	3	2697	0.11
	8ee9b856	0	2697	0.00
	d4406a1b	0	2697	0.00
	687a4eaf	1693	2697	62.77
	06a76847	0	2697	0.00
	15be6f61	4	2699	0.14

a large number of revisions and a low proof selection percentage that is not appropriate to be visualized with a bar chart. Table 2.2, and Table 2.3 show the results; the tables contain two parts, and we discuss each part in turn.

The top part of the tables show result summaries for each project; the sum and the average values are computed across 24 revisions. In Table 2.2,

the third column shows the number of proofs selected by iCoq and the fourth column shows the total number of proofs at each revision; the fifth column shows the proof selection percentage. For example, for Verdi, we find that iCoq executes a total of 4,458 proofs, while the existing technique executes 65,413 proofs across the same set of revisions. In other words, across all revisions, the proof selection percentage for iCoq is 7%. Note that the proof selection percentage is the same regardless of the execution environment.

The bottom part of the table shows detailed results for Verdi. We show the values for each revision; the revision SHA is shown in Column 2.

RQ2: We used the three large verification projects not only to obtain a proof selection percentage but also to obtain the proof checking time. First, we consider the CI-Env development environment. Recall that in CI-Env, `coq_makefile` will always execute all proofs and thus be costly. On the other hand, iCoq saves time by only running a subset of all proofs. Table 2.3 shows the proof checking time. Columns 3 and 4 show the proof checking time for CI-Env when using `coq_makefile` and iCoq, respectively. Table 2.4 shows the summaries. In summary, iCoq reduces the proof checking time 2.92×, 3.44×, and 9.62× for Flocq, UniMath, and Verdi, respectively. Note that CI-Env is of the highest importance due to the proliferation of CISs.

Although we also measured proof checking time for micro-benchmarks, we find that the time savings are insignificant in those cases due to very fast proof checking. Similar to regression test selection tools, which inspired our work, we believe that iCoq will be most beneficial to large verification projects

Table 2.3: Total and Average Proof Checking Time for Large Verification Projects using `coq_makefile` (Coq) and iCoq .

Project		**CI-Env Time [s]**			**LO-Env Time [s]**		
		Coq	**iCoq**	**c/i**	**Coq**	**iCoq**	**c/i**
Flocq	$\sum$	888.36	303.71	N/A	297.97	261.62	N/A
	Avg.	37.01	12.65	N/A	12.41	10.90	N/A
UniMath	$\sum$	12882.46	3742.88	N/A	3783.52	1692.33	N/A
	Avg.	536.76	155.95	N/A	157.64	70.51	N/A
Verdi	$\sum$	32528.57	3379.37	N/A	8157.45	3130.96	N/A
	Avg.	1355.35	140.80	N/A	339.89	130.45	N/A
	Revision	**Coq**	**iCoq**	**c/i**	**Coq**	**iCoq**	**c/i**
Verdi (details)	40d0e96f	1350.26	1375.29	0.98	1355.75	1375.88	0.98
	6b8a7d06	1351.02	63.38	21.31	1.07	5.58	0.19
	56b15cb5	1353.14	3.47	389.84	0.19	4.00	0.04
	9403f6f5	1351.62	148.83	9.08	1347.49	146.63	9.18
	112b39b0	1353.29	3.78	357.44	0.17	4.26	0.04
	57cf9bb1	1352.39	3.72	363.15	0.18	4.47	0.04
	bbf66a54	1349.02	3.71	363.03	0.18	4.43	0.04
	46b6be65	1352.35	3.88	348.36	0.18	4.83	0.03
	27537ec2	1352.00	3.68	366.99	0.19	4.08	0.04
	0f2b8090	1353.03	3.59	376.88	0.17	4.24	0.04
	0201fc23	1353.01	3.62	373.65	0.20	4.43	0.04
	cad0e753	1353.31	3.82	353.43	0.19	4.40	0.04
	2cb92f55	1350.86	147.53	9.15	1346.42	147.20	9.14
	21f660c1	1349.63	64.93	20.78	1351.98	10.11	133.60
	c28a126c	1350.41	3.80	355.09	0.19	6.15	0.03
	57479554	1351.44	64.94	20.81	6.19	8.53	0.72
	ade568dc	1345.22	3.61	372.22	0.20	4.10	0.05
	997ad0a6	1351.46	3.38	399.72	0.21	4.48	0.04
	cee72d1e	1346.28	65.00	20.71	6.14	8.57	0.71
	8ee9b856	1352.78	3.64	371.54	0.17	4.10	0.04
	d4406a1b	1349.74	3.55	379.46	0.20	4.11	0.05
	687a4eaf	1359.52	1178.12	1.15	1310.31	1169.79	1.12
	06a76847	1383.72	3.51	393.32	0.19	4.06	0.04
	15be6f61	1413.07	216.59	6.52	1429.29	192.53	7.42

with many dependencies and elaborate proofs.

RQ3: We were curious what savings could be obtained with iCoq in the LO-Env development environment. As when obtaining our answer to the previous question, we measured the proof checking time for large verification projects. Columns 6 and 7 in Table 2.3 show time for `coq_makefile` and iCoq, respec-

Table 2.4: Ratio of Total Times from Table 2.3.

Project	**CI-Env**	**LO-Env**
Flocq	2.92	1.13
UniMath	3.44	2.23
Verdi	9.62	2.60

tively. We can see that `coq_makefile` can save some proof checking time in LO-Env, i.e., whenever changes do not affect code. However, even if a change has minimal effect on code (e.g., in revision 9403f6f5 for Verdi), `coq_makefile` runs (almost) all proofs. We find (Table 2.4) that ICoq reduces the proof checking time 1.13×, 2×, and 3× on average, for Flocq, UniMath, and Verdi, respectively.

We believe the greater reduction in proof checking time for Verdi is primarily due to its many long-running proofs and opaque constants (that end in `Qed`.). In contrast, UniMath contains many non-opaque constants whose processing cannot be deferred during quick compilation, and nearly all proofs in Flocq have a relatively short running time.

2.5 Discussion

Safety: In a regression testing context, a test selection technique is *safe* when, for every possible change to a project, the technique never omits to run a test affected by the change [127]. Analogously, a proof selection technique is safe whenever no necessary proof checking task is ever omitted. ICoq currently

gives no formal guarantee of safety in this sense; a proof of safety would have to reason about Coq's toolchain, which is certainly possible at an abstract level, but difficult to do at the level of code. Nevertheless, verifying safety for a proof selection algorithm for Coq and Gallina is arguably more straightforward than doing so for a test selection algorithm for an object-oriented language with elaborate semantics (e.g., Java), which may include complicated features such as dynamic class loading.

Tactic language dependencies: ICoq currently does not perform parsing and dependency analysis of custom tactics defined in the Ltac language that occur in source files. This means that an isolated change in the definition of a tactic never results in lemmas whose (unedited) proof scripts contain calls to that tactic being marked as "changed", even though the semantics of such a proof script may have changed. Analysis of Ltac definitions is a planned future extension of ICoq. Similar concerns as for Ltac hold for custom Coq language extensions written in OCaml that are used in some projects.

Universe constraints: Sozeau and Tabareau recently introduced support for *generic* Coq definitions that can be used across universes of types [129]. However, Coq's toolchain for asynchronous proof processing ignores universe constraints, since such constraints must be checked for consistency at the global level [133]. Consequently, Coq projects that make heavy use of universe polymorphism are not good targets for ICoq.

Parameterized modules: A Coq *module* encapsulates a collection of definitions and lemmas in a namespace. A parameterized module, or *functor*, takes

modules with a certain signature as input, and can contain lemmas involving types in its parameters. Consequently, the file that contains the functor has corresponding proof tasks for those lemmas. However, no identifiers are exposed at the global level until the functor is fully instantiated with argument modules, eluding `coq-ast`. This problem can be solved, e.g., by conservatively compiling the file to a `.vo` file, checking all proofs. However, functors appear to be used rarely outside of the standard library; of the projects under study only Verdi uses them, and in a minimalistic way. Hence, we omitted support for functors in the initial version of ICoq.

Overhead: ICoq introduces several sources of overhead compared to LCF-style top-down processing of `.v` files into `.vo` files. One source is quick compilation and task-based proof checking itself, which is performed in independent phases and requires book-keeping for lemmas and proofs. Additionally, ICoq requires computing a fine-grained dependency graph and checksums to discover the impact of changes to a development. Consequently, ICoq may not be suitable to use in small-scale Coq projects, since the overhead can make regression proof selection as a whole take longer to complete than straightforward compilation to `.vo` files; similar conclusions were drawn for regression test selection [69].

2.6 Threats to Validity

External: Our results may not generalize to all Coq projects. To mitigate this threat, we used several micro-benchmarks and three large projects. The large

projects use different feature sets of Coq and target verification of disparate application domains. We used 24 revisions per project (for large projects), from segments in the revision histories with active development that were straightforward to compile with Coq version 8.5, the first version with asynchronous proof-checking support and the stable version available when we started development of ICoq. Our findings could differ for longer sequences of revisions and different segments in software histories. The number of revisions was determined by the setup cost and common practices in recent studies of regression testing techniques [69].

Internal: Our implementation of ICoq, as well as our evaluation infrastructure, may contain bugs. To mitigate this threat, we did extensive testing of our code and code reviews. In particular, we tested ICoq on a benchmark set of pairs of revisions of small Coq developments representing typical changes to proofs and definitions.

Construct: We implemented proof selection only for a single proof assistant (Coq). Although our technique should be applicable to other proof assistants (e.g., Isabelle/HOL), further work is needed to confirm the applicability.

2.7 Summary

We presented a technique for regression proof selection in large-scale verification projects, and its implementation for the Coq proof assistant in the tool ICoq. In particular, ICoq is suitable for use in continuous integration systems to quickly find failing proofs in rapidly evolving projects. By tracking

fine-grained dependencies, ICoq avoids checking unaffected proofs as changes are made to files. Our evaluation shows that using ICoq is up to 10× faster than checking all proofs from scratch (which is typical in a CI setting). ICoq can also be used from the command line, as an alternative to the default Makefile-based toolchain; our evaluation shows that ICoq is up to 3× faster in this case. While our implementation is Coq-specific, our technique works in any setting where it is possible to separate the processing of source files with proofs scripts into a fast pre-processing phase and a mostly independent, potentially time-consuming proof-checking phase.

Chapter 3

PICOQ: Parallel Regression Proving for Large-Scale Verification Projects[1]

Large-scale verification projects using proof assistants typically contain many proofs that must be checked at each new project revision. While proof checking can sometimes be parallelized at the coarse-grained file level to save time, recent changes in some proof assistant in the LCF family, such as Coq, enable fine-grained parallelism at the level of proofs. However, these parallel techniques are not currently integrated with regression proof selection (Chapter 2), a technique that checks only the subset of proofs affected by a change. In this chapter, we present techniques that blend the power of parallel proof checking and selection to speed up regression proving in verification projects, suitable for use both on users' own machines and in workflows involving continuous integration services. We implemented the techniques in a tool, PICOQ, which supports Coq projects. PICOQ can track dependencies between files, definitions, and lemmas and perform parallel checking of only those files or proofs affected by changes between two project revisions. We ap-

[1]Parts of this chapter are published at ISSTA 2018 [120]. My contributions span all aspects of this work: defining the project, designing a solution, implementing a tool, performing evaluation, and documenting findings.

plied PICOQ to perform regression proving over many revisions of several large open source projects and measured the proof checking time. While gains from using proof-level parallelism and file selection can be considerable, our results indicate that proof-level parallelism and proof selection is consistently much faster than both sequential checking from scratch and sequential checking with proof selection. In particular, 4-way parallelization is up to 28.6 times faster than the former, and up to 2.8 times faster than the latter.

3.1 Overview

Many large-scale verification projects rely on *proof assistants* to construct and check formal proofs [67]. Representative projects target critical software domains, e.g., compilers [103], operating systems [95], file systems [31], and distributed systems [105, 151], or formalize mathematical theories [22, 74].

Results certified by proof assistants are highly trustworthy, but establishing properties demands significant time investment by sophisticated users to guide the proof effort. When such projects are modified, previously proven properties must be *reestablished*, since even small changes can break a critical step in a proof—this process of *regression proving* may take anywhere from seconds to tens of minutes [28], and in extreme cases, several days [82].

Analogously to when building and testing software engineering projects, productivity suffers in verification projects when regression proving takes an inordinate amount of time. One important technique for speeding up software builds and tests on commodity multi-core hardware is *parallelization* [12, 27,

97]. Due to recent changes in popular proof assistants in the LCF family, such as Coq [9] and Isabelle/HOL [147], parallelization of proof checking is now possible not only at the coarse-grained level of *files* (via build systems such as `make`) but also at the fine-grained level of individual *proofs*. As we have argued in the previous chapter, a formal proof of some program property can be viewed as representing many (possibly an infinite number of) tests. Based on this analogy, coarse-grained parallel proof checking intuitively corresponds to test suite parallelization at the *test class* level in Java-like languages, while fine-grained parallel proof checking corresponds to parallelization at the *test method* level.

Coarse-grained parallel proof checking is used in many Coq verification projects, e.g., when building such projects via the OCaml Package Manager (OPAM) [50,116]. One important reason for widespread use of coarse-grained parallelism is that proof checking using proof assistants in the LCF family is deterministic and occurs in a predictable runtime environment. Additionally, when surveying 260 publicly available Coq projects on GitHub, each with more than 10k lines of code (LOC), we found that ∼20% of these projects can also leverage fine-grained parallel proof checking, two years after support was added to Coq.

Regression proof selection (Chapter 2), a technique that avoids checking proofs unaffected by changes as a project evolves, is orthogonal to parallel proof checking. Unfortunately, at present, regression proof selection is not integrated with parallel proof checking. This means that large-scale projects

must generally perform regression proving from scratch, in particular when using (as they often do) continuous integration services (CISs) such as Travis CI [137]. Even with parallelism using many cores, the resulting long proof checking time can be a burden to users.

In this chapter, we describe techniques that blend proof checking parallelization and selection to speed up regression proving in large-scale verification projects. As we demonstrate in our evaluation, these novel techniques are superior to *legacy* (state-of-the-art) techniques even on users' own machines, but are particularly effective when used in CISs. We believe that our techniques can alleviate the cost to productivity and trust in evolving large-scale verification projects caused by long proof checking times [4], and release untapped potential in multi-core hardware for regression proving.

This chapter describes the following contributions:

⋆ **Techniques**: We propose novel techniques that integrate parallel proof checking and selection to speed up regression proving in evolving verification projects using proof assistants. Along one axis, we consider coarse-grained and fine-grained proof checking parallelism. Along the other axis, we consider selection at both the file and proof levels, i.e., we check only those files or proofs that are affected by changes. The result is a *taxonomy* of regression proving techniques that also includes legacy techniques.

⋆ **Tool**: We implemented our techniques in a tool, dubbed PICOQ, which supports Coq projects. PICOQ relies on a collection of extensions to the Coq

proof checking toolchain, several of which originate in the iCoq tool [29]. We provide a version of piCoq on the following URL: `http://cozy.ece.utexas.edu/icoq`.

⋆ **Evaluation**: We performed an empirical study to measure the effectiveness of our regression proving techniques using piCoq. We used many revisions of several large-scale open source Coq projects, and measured the proof checking times for our techniques and legacy techniques. Our results show that speedups can be substantial from adopting proof-level parallelism and file selection, but that improvements vary significantly across projects, and may even be absent. However, combined proof-level parallelism and proof selection is consistently much faster than both sequential checking from scratch (legacy) and sequential checking with proof selection (iCoq). Specifically, in a CIS environment, we obtain a speedup of up to 28.6× with 4-way fine-grained parallelism and proof selection compared to the former, while the speedup compared to the latter is up to 2.8×.

3.2 Coq Background

The Coq proof assistant consists of, on the one hand, an implementation of a small and powerful purely functional programming language, and on the other hand, a system for specifying properties about programs and proving them. Coq is based on a type theory called the Calculus of Inductive Constructions [123], where both programs and propositions about programs are *types* inhabited by *terms*. In effect, this property puts program development

and proving on the same footing for Coq users.

In a typical workflow for a Coq-based project, users interactively construct tentative proof terms for propositions using operations called *tactics*. Propositions are only accepted as proven after Coq's type checker has been run successfully on a proof term. Absent inconsistent axioms and frontend issues, the user need only trust that the comparatively small type checking kernel is correctly implemented and compiled to trust that proven propositions really hold. Figure 3.1 illustrates the interactive proof development process.

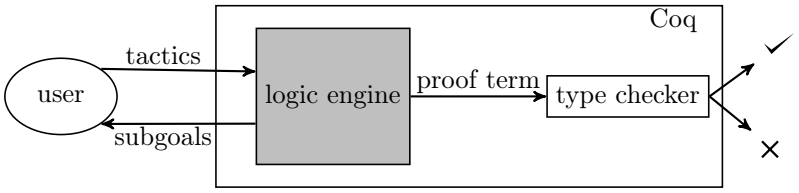


Figure 3.1: Coq interactive proof development overview.

3.2.1 Coq Proof Checking Toolchain

Definitions of functions and propositions processed by Coq are written in the Gallina language, and reside in files ending in `.v`. The Coq batch proof checking (compilation) tool, `coqc`, takes a `.v` file as input and, by default, produces a `.vo` file as output that contains full binary representations of processed Gallina constructs, including proofs. Since files may depend on other files, checking all proofs in a Coq project requires some form of dependency analysis. The standard `coq_makefile` tool generates a Makefile which, by default, calls the `coqdep` tool for this purpose [44]. `coqdep` builds a dependency graph for all input `.v` files based on simple syntactic analysis of

`Require` commands in files (similar to `import` statements in Java), which indicate direct dependency at the file level. When proof checking is performed via the generated Makefile, the dependency graph is used to process `.v` files with `coqc` in some allowed order. The Makefile also enables timestamp-based incremental regression proving in a Coq project, as well as spawning of parallel proof-checking processes. Note that such proof-checking parallelism is fundamentally restricted by the file dependency graph; for example, if this graph is a path (has no branches) there will be no parallel checking at all.

3.2.2 Asynchronous Proof Checking in Coq

Coq version 8.5, the first stable release to include architectural changes to support a *document-oriented* interaction model [9], introduced the option to *quick-compile* `.v` files to the binary `.vio` format, a process which avoids checking (and emitting representations of) proofs that are indicated as *opaque* by ending with `Qed`. Figure 3.2 illustrates the new `.vio` proof checking workflow made possible by Coq's document-oriented model. Only the type (proposition) of an opaque lemma, i.e., not the body proof term, can be referenced in other parts of a Coq development, whence type checking of all such terms can normally be performed in complete isolation, and thus in parallel. Specifically, `.vio` files contain *proof-checking tasks*, which can be performed individually 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 in most developments, but must then assume that all proofs are correct.

Asynchronous proof checking has two important applications in large-scale Coq projects. First, it enables *regression proof selection*, i.e., the possibility of checking only affected proofs after each new project revision [28, 58]. Second, it enables *fine-grained* parallel proof checking that can make better use of commodity multi-core hardware than file-level parallel checking [9, 147]. Specifically, `coqc` includes the option `-schedule-vio-checking`, which takes as arguments (i) an upper bound on the number of parallel processes, and (ii) a list of `.vio` files whose proof tasks to check in parallel. However, note that there is no way to specify *subsets* of proof tasks in files to check in parallel. In contrast to purely Makefile-based task parallelism, which often fails to utilize the requested number of parallel processes throughout proof checking due to file dependency restrictions, the degree of parallelism for fine-grained checking depends directly on how checking of individual proofs is scheduled on the parallel operating system processes.

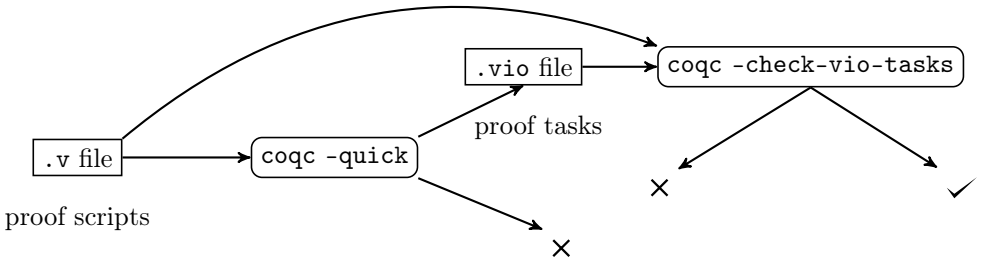


Figure 3.2: Coq asynchronous proof checking workflow.

Coq uses a notion of *sections* to organize common assumptions made in a collection of lemmas, say, that equality on a type `A` is decidable (`A_eq_dec`).

A lemma may reference one or more such assumptions, which then become quantified variables that must be instantiated when the lemma is referenced outside of the section. However, by default, Coq only determines the used section variables of a lemma when the end of the section is reached. This means that the final type (assertion) of the section lemma is not known when considered in isolation, whence its proof cannot be immediately checked as an asynchronous task. To get around this problem, Coq allows section lemmas to be annotated with the assumptions they use (e.g., `Proof using A_eq_dec`). The required annotations can be derived from metadata produced by Coq during compilation of source files to `.vo` files [133], and then inserted back into the source files. In the evaluation of our techniques (Section 3.6), we used this approach to automatically add annotations to all revisions of the projects under study.

3.2.3 iCoq and Regression Proof Selection

iCoq is a tool for (sequential) regression proof selection in Coq (Chapter 2). iCoq tracks dependencies among both files and proofs in order to check only those proofs affected by changes to a project, potentially saving significant time in comparison to checking everything from scratch. iCoq processes Coq projects in three phases (each similar to the corresponding phase in regression test selection tools [69, 118, 127]): *analysis*, *proof checking*, and *dependency collection*. In the analysis phase, iCoq detects files and proofs that are affected by changes made since the last run of iCoq. In the proof checking phase, iCoq uses Coq's toolchain for asynchronous processing to check only

the proofs selected in the analysis phase (but not other proofs). Finally, in the collection phase, ICoq obtains the new dependencies that will be used in the next run of the analysis.

3.3 Running Example

We use the small Coq library of list functions and lemmas shown in figures 3.3, 3.4, and 3.5 to illustrate our techniques; code is extracted from the StructTact project [131]. The Coq standard library contains a function `remove` that, when given a decision procedure for equality for a type `A`, removes a single element from a list of that type. The file `ListUtil.v` contains two lemmas about the `remove` function. `Dedup.v` defines a function `dedup` that omits any duplicates from the argument list, and a lemma about this function. `RemoveAll.v` defines a function `remove_all` that removes *all* elements identical to the given element from a list, and two lemmas about this function.

As indicated by the `Require` commands and by direct references inside proof scripts in `Dedup.v` and `RemoveAll.v`, the proofs of lemmas in these files depend on lemmas in `ListUtil.v`. For example, the proof of the lemma `remove_all_in` in `RemoveAll.v` depends on the lemma `in_remove` in the `ListUtil.v` file. Figure 3.6 shows both the file-level and proof-level dependencies of the project. File dependencies are illustrated by solid line arrows, and dependencies among definitions, lemmas, and proofs by dashed arrows. As indicated in the figure, the proofs of lemmas in both `Dedup.v` and `RemoveAll.v` depend on the utility lemmas in `ListUtil.v`, but not all of the former depend

```
Require Import List. Import ListNotations.

Lemma remove_preserve : ∀ (A : Type) A_eq_dec (x y : A) xs,
  x ≠ y → In y xs → In y (remove A_eq_dec x xs).
Proof.
induction xs; simpl; intros.
- intuition.
- case A_eq_dec; intros.
  * apply IHxs; subst; intuition.
  * intuition; subst; left; auto.
Qed.

Lemma in_remove : ∀ (A : Type) A_eq_dec (x y : A) xs,
  In y (remove A_eq_dec x xs) → In y xs.
Proof.
induction xs; simpl; intros; auto.
destruct A_eq_dec; simpl in *; intuition.
Qed.
```

Figure 3.3: `ListUtil.v` from an example Coq project.

on all of the latter. Nevertheless, the default Coq proof-checking toolchain checks all proofs and writes `.vo` files whenever a change is made to some utility lemma in `ListUtil.v`.

In contrast, the asynchronous proof-checking toolchain, complemented by a tool such as ICoq, allows avoiding many instances of proof checking when making changes to `ListUtil.v`. For example, suppose the maintainers of the project change the definition of the lemma `in_remove` to the one in Figure 3.7 where variables `x` and `y` are swapped (highlighted). To ensure that all previously proven properties in the library hold, the proofs of `remove_dedup` and `remove_all_in` both need to be checked in addition to `in_remove`. To perform these tasks, we first compile the `.v` files to `.vio` files (which elides checking of all lemmas ending in `Qed`). Then, we issue individual proof-checking

```
Require Import List ListUtil. Import ListNotations.

Fixpoint dedup (A : Type) A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  if in_dec A_eq_dec x xs then dedup A A_eq_dec xs
  else x :: dedup A A_eq_dec xs
end.

Lemma remove_dedup : ∀ A A_eq_dec (x : A) xs,
  remove A_eq_dec x (dedup A A_eq_dec xs) =
  dedup A A_eq_dec (remove A_eq_dec x xs).
Proof.
induction xs; intros; auto; simpl.
repeat (try case in_dec; try case A_eq_dec; simpl; intuition);
  auto using f_equal.
- exfalso. apply n0. apply remove_preserve; auto.
- exfalso. apply n. apply in_remove in i; intuition.
Qed.
```

Figure 3.4: `Dedup.v` from an example Coq project.

commands separately for the three lemmas, which reveals that their proofs can be reestablished.

As another example, consider the proposed change (highlighted) to the `dedup` function in Figure 3.8. Intuitively, the change only removes some code duplication by using a `let` expression to encapsulate the recursive call, i.e., the meaning of the function is preserved. Since this change only affects `Dedup.v`, this is the only file that we need to recompile to a `.vo` file when relying on the default toolchain (assuming a recent `ListUtil.vo` file is available). With the asynchronous proof-checking toolchain, we need to recompile `Dedup.v` to a `.vio` file, and then issue a proof checking command for the lemma `remove_dedup`.

```
Require Import List ListUtil. Import ListNotations.

Fixpoint remove_all A A_eq_dec (to_delete l : list A) : list A :=
match to_delete with
| [] ⇒ l
| d :: ds ⇒ remove_all A A_eq_dec ds (remove A_eq_dec d l)
end.

Lemma remove_all_in : ∀ A A_eq_dec ds l x,
  In x (remove_all A A_eq_dec ds l) → In x l.
Proof.
induction ds; simpl; intros; intuition.
eauto using in_remove.
Qed.

Lemma remove_all_preserve : ∀ A A_eq_dec ds l x,
  ~ In x ds → In x l → In x (remove_all A A_eq_dec ds l).
Proof.
induction ds; simpl; intros; intuition auto using remove_preserve.
Qed.
```

Figure 3.5: `RemoveAll.v` from an example Coq project.

3.4 Techniques

This section describes our taxonomy of regression proving techniques, which includes both legacy techniques and our proposed novel techniques. To concretize the presentation, we describe the techniques as *proof checking modes* for Coq, as defined in Table 3.1.

A fundamental choice when checking Coq proofs is whether to use default compilation or quick-compilation of source files. Opting for default compilation means that all proof-checking must be performed top-down according to the file-level dependency graph, which also restricts the (file-level) parallelism according to this graph. With default compilation, a user can either

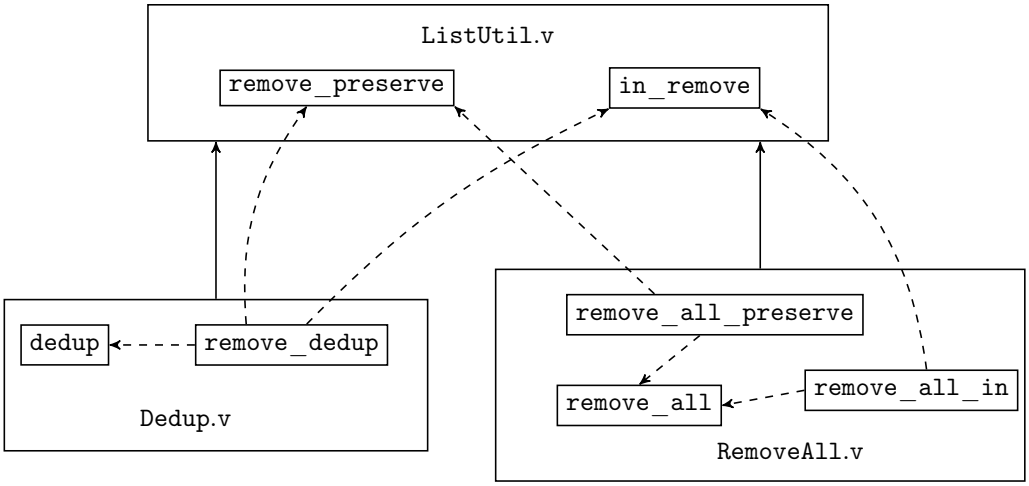


Figure 3.6: Dependencies for the example Coq project shown in figures 3.3, 3.4, and 3.5.

```
Lemma in_remove : ∀ (A : Type) A_eq_dec (x y : A) xs,
  In x (remove A_eq_dec y xs) → In x xs.
Proof.
induction ys; simpl; intros; auto.
destruct A_eq_dec; simpl in *; intuition.
Qed.
```

Figure 3.7: Revised version of lemma `in_remove` in the file `ListUtil.v` in Figure 3.3, with changed line highlighted.

perform *no selection* (`f•none`), i.e., check the whole project from scratch, or coarse-grained *file-level selection* (`f•file`), where only proofs in files affected by actual changes are checked. With quick-compilation, the previous two forms of selection (`p•none` and `p•file`) are complemented by fine-grained *proof-level selection*, where only individual proofs affected by changes are checked (`p•icoq`).

We consider two execution environments for each proof checking mode: CI-Env and LO-Env. CI-Env describes an environment that uses a Continuous

```
Fixpoint dedup (A : Type) A_eq_dec (xs : list A) : list A :=
match xs with
| [] ⇒ []
| x :: xs ⇒
  let tail := dedup A A_eq_dec xs in
  if in_dec A_eq_dec x xs then tail else x :: tail
end.
```

Figure 3.8: Revised version of function `dedup` in the file `Dedup.v` in Figure 3.4, with changed lines highlighted.

Table 3.1: Modes for Regression Proving in Coq.

Parallelization	**Selection**		
Granularity	*None*	*Files*	*Proofs*
File level	`f·none`	`f·file`	N/A
Proof level	`p·none`	`p·file`	`p·icoq`

Integration Service (CIS) [85], e.g., Travis CI, to check proofs. Note that a CIS checks proofs in a clean environment for each revision. LO-Env describes an environment where developers use their local machines to check proofs. Note that file timestamps and generated files, not present in version control, are preserved in LO-Env, but not in CI-Env.

We next describe the details of each mode and discuss variants of each mode for CI-Env and LO-Env.

`f·none`: This legacy mode embodies the approach used in the default Coq proof-checking toolchain with `coq_makefile`. Since all files are fully checked for every revision, there is no difference between running this mode in LO-Env and CI-Env. Many large-scale projects on GitHub use this mode in their

Table 3.2: `f•none` Mode for Coq Project Shown in Figures 3.3, 3.4, and 3.5; Same-Phase Tasks Can Run in Parallel.

Phase	Task	Definitions and Lemmas
1	ListUtil.vo	remove_preserve, in_remove
2	Dedup.vo	dedup, remove_dedup
2	RemoveAll.vo	remove_all, remove_all_in, remove_all_preserve

Travis CI jobs, e.g., Verdi [151]; we therefore used it as the CIS baseline when investigating the speedup from sequential proof selection using ICoq [28].

On one hand, this mode has no overhead from proof checking task management and tracking dependencies across revisions. On the other hand, parallelism is restricted by the file dependency graph, and there is overhead from writing (possibly large) `.vo` files to disk. Table 3.2 illustrates how parallel checking can be performed using the `f•none` mode for the example project shown in figures 3.3, 3.4, and 3.5.

`p•none`: This legacy mode embodies the approach used in the asynchronous proof-checking toolchain introduced by Barras et al. [9]. As in `f•none`, which is the closest comparable alternative, there is no difference between running the mode in LO-Env and CI-Env. Although `coq_makefile` generates tasks to use this mode, we found that ∼20% of 260 projects on GitHub that we analyzed (with more than 10k LOC each) can use `p•none` properly without modification. The reason for this percentage not being higher is the requirement to annotate proofs inside sections, as explained in Section 3.2.

On one hand, proof checking in `p•none` is not restricted by the file de-

Table 3.3: `p•none` Mode for Coq Project Shown in Figures 3.3, 3.4, and 3.5; Same-Phase Tasks Can Run in Parallel and Proof Tasks (to be Run in a Later Phase) are in Bold.

Phase	Task	Definitions and Lemmas
1	ListUtil.vio	**remove_preserve**, **in_remove**
2	Dedup.vio	dedup, **remove_dedup**
2	RemoveAll.vio	remove_all, **remove_all_in**, **remove_all_preserve**
3	checking	remove_preserve
3	checking	in_remove
3	checking	remove_dedup
3	checking	remove_all_in
3	checking	remove_all_preserve

pendency graph, and there is no overhead from writing proof terms to disk or tracking dependencies across revisions. On the other hand, this mode requires `.vio` file compilation of (annotated) `.v` files according to the file dependency graph, and has overhead from managing individual proof checking tasks. Table 3.3 illustrates the maximum possible parallelism of this mode for the project shown in figures 3.3, 3.4, and 3.5. In the table, lemmas whose proofs are exempt from checking in a task are marked in bold. Note that the possible degree of parallelism is greater than for `f•none` due to isolated checking of proofs. The degree of parallelism can be adjusted downwards by moving proofs from one checking task to another.

`f•file`: This novel mode adds file-level selection to `f•none` by only compiling *affected* `.v` files to `.vo` files between revisions, with directly modified files determined by comparing current file checksums to previous checksums. As such, `f•file` suffers from the overhead of maintaining a file dependency graph using

`coqdep`, but not from managing proof-checking tasks. In LO-Env, this mode corresponds to the baseline we compared sequential proof selection against in the previous chapter using ICoq, which modeled developers incrementally checking projects on their local machines. In CI-Env, dependency graph and file checksum metadata must be explicitly persisted. Moreover, additional `.vo` files (those on which modified files depend) may have to be compiled in CI-Env compared to LO-Env for the same change to a project, since previously compiled `.vo` files are not available. Same as for `f•none`, the degree of parallelism is restricted by project file dependencies, and all proof terms in selected files are written to disk.

Suppose we are working with the project shown in figures 3.3, 3.4, and 3.5 and perform the change indicated in Figure 3.7. In `f•file` for both LO-Env and CI-Env, regression proving would then entail (re)compiling all `.v` files to `.vo` files, with parallelism as in Table 3.2. If we instead perform the change in Figure 3.8, `f•file` in LO-Env only recompiles `Dedup.v` into `Dedup.vo`, without any possibility of parallelism. In CI-Env, both `ListUtil.v` and `Dedup.v` are compiled into `.vo` files (sequentially).

`p•file`: This novel mode adds file-level selection to `p•none`, using the same analysis of file changes and file dependencies as in `f•file`. Consequently, `p•file` has overhead both from maintaining a file dependency graph and for management of proof-checking tasks, but is not restricted by file dependency graph for proof-checking parallelization. After determining and quick-compiling the necessary files (using file checksums), the mode uses the `coqc`

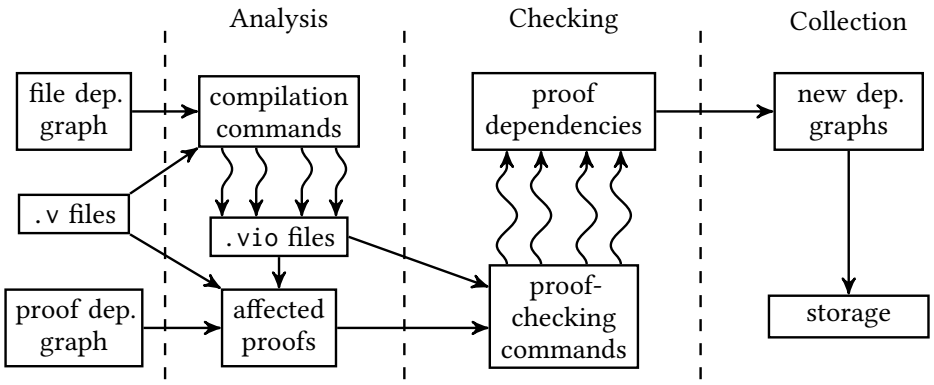


Figure 3.9: `p·icoq` workflow/phases with 4-way parallelism.

command `-schedule-vio-checking` described in Section 3.2.2. Running this command will typically check many unaffected proofs needlessly. However, `p·file` does not write proof terms to disk.

Suppose we are working with the project shown in figures 3.3, 3.4, and 3.5 and perform the change indicated in Figure 3.7. In `p·file` for both LO-Env and CI-Env, regression proving would then entail recompiling all `.v` files to `.vio` files and then reestablishing all lemmas in all `.vio` files, via the same parallelism as in Table 3.3. If we instead perform the change in Figure 3.8, `p·file` in LO-Env entails recompiling `Dedup.v` into `Dedup.vio`, and then checking all proof tasks in that file, i.e., checking `remove_dedup` asynchronously. `p·file` in CI-Env needs to compile both `ListUtil.v` and `Dedup.v` into `.vio` files, but runs the same `coqc` command to check `remove_dedup`.

`p·icoq`: This novel mode, which is the most sophisticated one in our taxonomy, combines fine-grained parallelism with proof selection, and corresponds

Table 3.4: `p•icoq` Mode for Change Shown in Figure 3.7 to Project Shown in Figures 3.3, 3.4, and 3.5; Same-Phase Tasks Can Run in Parallel, and Proof Tasks (to be Run in a Later Phase) are in Bold.

Phase	Task	Definitions and Lemmas
1	ListUtil.vio	**remove_preserve**, **in_remove**
2	Dedup.vio	dedup, **remove_dedup**
2	RemoveAll.vio	remove_all, **remove_all_in**, **remove_all_preserve**
3	checking	in_remove
3	checking	remove_dedup
3	checking	remove_all_in

to ICoq when used sequentially (i.e., with one proof-checking process). The mode incurs overhead both from proof task management and from tracking of proof-level dependencies. However, it features proof checking parallelism unrestricted by file dependencies and can elide checking unaffected proofs regardless of their location in a file. Moreover, no proof terms are written to disk. In both LO-Env and CI-Env, `p•icoq` relies on file checksumming to first locate changed files, which are then subject to more detailed impact analysis at the level of proofs. Figure 3.9 illustrates the workflow for `p•icoq` in the case of four parallel jobs for quick-compilation and fine-grained proof checking.

Suppose we are working with the project shown in figures 3.3, 3.4, and 3.5 and perform the change indicated in Figure 3.7. In `p•icoq` for both LO-Env and CI-Env, regression proving would first entail recompiling all `.v` files to `.vio` files, and then proving asynchronously `in_remove`, `remove_dedup`, and `remove_all_in` (and skipping `remove_preserve` and `remove_all_preserve`, which are unaffected). Parallelism for this case is illustrated in Table 3.4. If we

instead perform the change in Figure 3.8, `p•icoq` in LO-Env entails recompiling `Dedup.v` into `Dedup.vio`, and then checking `remove_dedup` asynchronously. In contrast, CI-Env regression proving requires quick-compiling both `ListUtil.v` and `Dedup.v`, and then checking `remove_dedup`.

3.5 Implementation

We implemented the modes described in Section 3.4 in a tool dubbed PICOQ, written in OCaml, Java, and Bash. Since Coq developments are not upwards or downwards compatible in general, we target Coq version 8.5 to support the largest range of project revision histories susceptible to asynchronous proof checking and fine-grained parallelism; we expect no fundamental issues with supporting future Coq versions.

We extended `coqc` (the official Coq compiler) with the new option `-schedule-vio-task-depends-checking`. As first argument, it takes an upper bound on the number of parallel processes, and then a list of proof task definitions (pairs of `.vio` file names and task identifiers). `coqc` with the new option (`-schedule-vio-task-depends-check-ing`) performs parallel proof checking of all indicated tasks in the same way as the official `-schedule-vio-checking` option (which only takes whole files as arguments), but also outputs the dependencies of each processed proof individually. This dependency data can then be used by PICOQ to select affected proofs in the next revision.

We rely on the same Coq plugins and extensions as ICOQ, adapted for parallel proof checking. We also use the graph-based analysis from ICOQ

Table 3.5: Projects Used in the Evaluation.

Project	**URL**	**SHA**	**LOC**	**#Revs**	**#Files**	**#Proof Tasks**
Coquelicot	[43]	680ca587	38260	24	29	1660
Finmap	[59]	baec7ba0	5661	23	4	959
Flocq	[60]	4161c990	24786	23	40	943
Fomegac	[61]	7a654d7c	2637	14	13	156
Surface Effects	[132]	3450e4b7	9621	24	15	289
Verdi	[141]	15be6f61	56147	24	222	2756
$\sum$	N/A	N/A	137112	132	323	6763
Avg.	N/A	N/A	22852.00	22.00	53.83	1127.16

to find affected files and proofs across revisions, which is similar to the approach used in build systems such as Google's Bazel [10] and Microsoft's CloudMake [57]. We use Java's task executor facilities [72] for parallel compilation of `.vo` and `.vio` files via `coqc` commands.

3.6 Evaluation

To assess the efficacy of our proposed techniques on large, evolving verification projects, we answer the following research questions:

RQ1: How effective, in terms of proof checking time, is coarse-grained parallel regression proving (file-level parallelism) without any selection, i.e., `f·none`?

RQ2: How effective, in terms of proof checking time, is fine-grained parallel regression proving (proof-level parallelism) without any selection, i.e., `p·none`?

RQ3: How effective, in terms of proof checking time, is coarse-grained parallel regression proving with selection at the level of files using PICOQ, in CISs and

on developers' own machines, i.e., `f·file` in CI-Env and LO-Env?

RQ4: How effective, in terms of proof checking time, is fine-grained parallel regression proving with selection at the level of files using PICOQ, in CISs and on developers' own machines, i.e., `p·file` in CI-Env and LO-Env?

RQ5: How effective, in terms of proof checking time, is fine-grained parallel regression proving with selection at both the level of files and individual proofs using PICOQ, in CISs and on developers' own machines, i.e., `p·icoq` in CI-Env in LO-Env?

We run all experiments on a 4-core Intel Core i7-6700 CPU @ 3.40GHz machine with 16GB of RAM, running Ubuntu 17.04. We limit the number of parallel processes to be at or below the number of physical cores; this avoids the problem of drawing clear conclusions about speedups when using virtual cores (hyper-threading).

3.6.1 Verification Projects Under Study

Table 3.5 shows the list of Coq projects used in our study; all projects are publicly available. We selected projects based on (a) public availability of their revision history during principal development, (b) compatibility of their revision history with Coq version 8.5, (c) their size and popularity, and (d) tractability of their build process; the latter was necessary for a successful experimental setup. For each project, we show the name, the repository URL, the last revision/SHA we used for our experiments, the number of lines of Coq code (LOC) for the last revision, as reported by `cloc` [38], the number

of revisions that we considered, the number of `.v` files in the project, and the number of proof tasks in the project.

Note that since Coq projects have different development paces and added support for Coq 8.5 at different points in time, our revision ranges are not all from the same time period.

Coquelicot: Coquelicot is a library for real number analysis [22], containing results about limits, derivatives, integrals, etc.

Finmap: Finmap is a library of definitions and results about finite sets and finite maps [59], based on the Mathematical Components library [52, 75].

Flocq: Flocq is a library that formalizes floating-point arithmetic in several representations [21], e.g., as described in the IEEE-754 standard. Flocq is used in the CompCert verified C compiler to reason about programs which use floating-point operations [20].

Fomegac: This project contains a formalization of a version of the formal system F_ω and the corresponding metatheory, such as type safety results [61].

Surface Effects: This project formalizes a functional programming language of "surface effects" with operations on mutable state [126], including its operational semantics and metatheory (typability, effect soundness and correctness).

Verdi and Verdi Raft: Verdi is a framework for verification of implementations of distributed systems [150]. While the framework is not currently tied to any one particular verification project, it was initially bundled with a verified implementation of the Raft distributed consensus protocol [151]. Each revision

```
Require: p a project under study
Require: κ the number of revisions
Require: ε a development environment
Require: η range of number of parallel jobs
 1: procedure ExperimentProcedure(p, κ, η, ε)
 2:     Clone(p.url)
 3:     for all ι ∈ {1, ..., η} do
 4:         for all ρ ∈ LatestRevisions(κ, p) do
 5:             Checkout(ρ)
 6:             Configure(p, ε, ι)
 7:             SelectExecuteAndCollect(p)
 8:         end for
 9:     end for
10: end procedure
```

Figure 3.10: Experiment procedure.

comprises over 50k LOC, making Verdi one of the largest publicly available software verification projects.

3.6.2 Variables

We manipulate three independent variables in our experiments: *proof checking mode*, *development environment*, and (maximum) *number of parallel jobs*. The proof checking modes and environments are as described in Section 3.4. The number of parallel jobs ranges from 1 to 4. As a dependent variable we consider only the proof checking time.

3.6.3 Experiment Procedure

Figure 3.10 shows our experiment procedure for collecting the data necessary to answer our research questions. The inputs to the procedure include one of the projects used in the study, number of revisions to use in the experiment, a development environment, and a range of number of parallel jobs. In

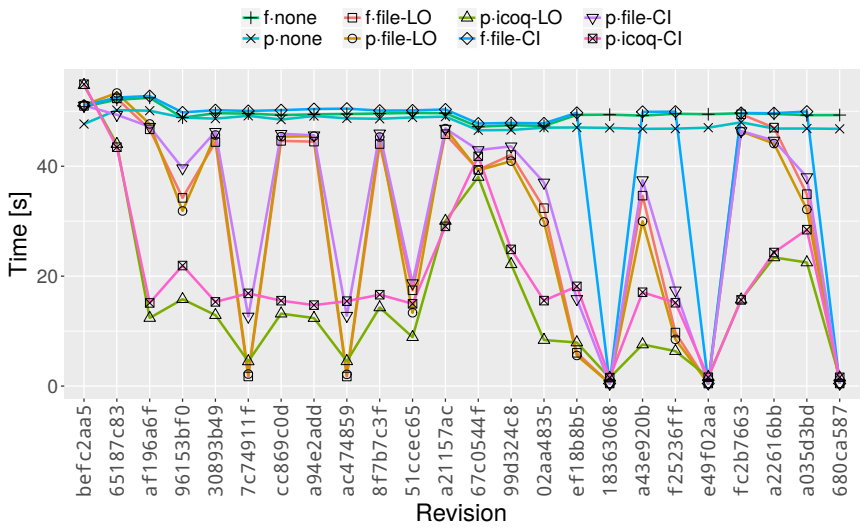
the initial step (line 2), the procedure clones the project repository from the URL in Table 3.5. Next, the procedure iterates over the range of parallel jobs, starting from one, until the upper bound η is reached. For a particular number of parallel jobs, the procedure iterates over κ revisions, from the oldest to the newest revision. In each iteration of the inner loop, the procedure (a) obtains a copy of the project for the current revision (line 5), (b) configures the project (in preparation for proof checking), and (c) selects proofs or files that are affected by changes and checks them. While executing the procedure, we log the time for each step of the procedure and the number of executed proofs; we report the former and use the latter to check correctness of our experiments.

It is important to observe that some PICOQ modes require persisting dependency metadata files between each revision. One way to do this in a CIS is to use built-in caching facilities [130]. Since the dependency data is small, we do not associate any overhead with persisting these files, even in CI-Env.

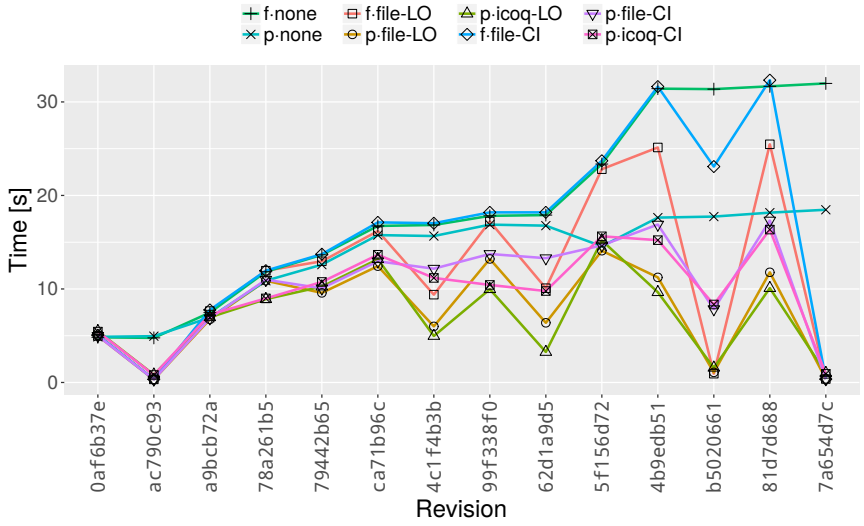
3.6.4 Results

We illustrate some of the data collected with our procedure in Figure 3.11. The plots (for Coquelicot and Fomegac) show, for every revision of the projects, the proof checking time in all modes when using 4 cores.

Table 3.6, and Table 3.7 list the total proof checking times for all projects and modes. The first column shows the name of the project, and the second column shows the name of the mode. Columns three to six show the results for each number of parallel jobs. Recall that parallel checking does



(a) Coquelicot



(b) Fomegac

Figure 3.11: Comparison of proof checking times for different modes across revisions of Coquelicot (top) and Fomegac (bottom). The plots show the proof checking time using four parallel jobs.

Table 3.6: Total Execution Time in Seconds of Projects Coquelicot, Finmap, and Flocq for All Modes and Different Number of Jobs.

Project	Mode	Time [jobs=1]	[jobs=2]	[jobs=3]	[jobs=4]
Coquelicot	f·none	1807.49	1226.33	1186.49	1187.29
	p·none	2319.39	1400.08	1260.69	1150.87
	f·file-LO	1043.01	745.41	724.32	725.55
	p·file-LO	1221.63	814.89	747.11	708.50
	p·icoq-LO	552.05	407.84	396.55	384.43
	f·file-CI	1587.52	1086.38	1049.53	1051.73
	p·file-CI	1405.10	885.67	802.13	786.88
	p·icoq-CI	732.60	528.78	505.49	479.79
Finmap	f·none	646.77	549.91	549.50	549.62
	p·none	1377.30	800.71	766.60	758.22
	f·file-LO	236.77	232.74	232.64	232.08
	p·file-LO	459.30	334.14	225.59	196.15
	p·icoq-LO	227.31	179.55	153.42	148.11
	f·file-CI	279.04	274.72	274.09	274.29
	p·file-CI	473.59	346.19	234.34	205.13
	p·icoq-CI	258.72	191.73	164.81	159.54
Flocq	f·none	892.54	525.44	516.65	512.46
	p·none	1173.08	702.60	614.19	579.45
	f·file-LO	319.22	197.82	190.47	189.68
	p·file-LO	390.66	244.64	227.00	216.38
	p·icoq-LO	274.63	194.55	183.71	175.62
	f·file-CI	370.22	230.27	223.21	221.15
	p·file-CI	430.79	260.61	233.30	231.80
	p·icoq-CI	320.37	216.64	206.25	196.08

not necessarily use all available parallel processes all the time, since tasks may depend on other tasks.

RQ1: For Verdi in `f·none`, going from sequential to 4-way parallel coarse-grained checking jobs brings a speedup of 3.3×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.5×, 1.2×, 1.7×,

Table 3.7: Total Execution Time in Seconds of Projects Fomegac, Surface Effects, and Verdi for All Modes and Different Number of Jobs.

Project	**Mode**	**Time**			
		[jobs=1]	**[jobs=2]**	**[jobs=3]**	**[jobs=4]**
Fomegac	f·none	278.31	261.13	261.37	261.66
	p·none	293.33	222.66	199.18	191.88
	f·file-LO	171.80	165.41	165.32	165.44
	p·file-LO	176.15	135.29	117.64	109.17
	p·icoq-LO	153.69	121.88	109.95	101.33
	f·file-CI	228.11	220.48	220.31	220.50
	p·file-CI	210.82	168.78	151.20	142.48
	p·icoq-CI	188.18	155.84	142.80	134.75
Surface Effects	f·none	8712.55	8633.68	8627.13	8629.28
	p·none	8465.24	4863.77	4067.83	3902.42
	f·file-LO	7820.60	7796.43	7781.69	7787.54
	p·file-LO	7599.11	4408.28	3432.82	3237.19
	p·icoq-LO	3669.35	2288.69	2033.12	2014.79
	f·file-CI	8714.85	8652.68	8651.51	8656.17
	p·file-CI	7695.15	4595.89	4047.15	3769.91
	p·icoq-CI	4116.04	2723.55	2404.59	2351.96
Verdi	f·none	35713.55	19157.52	13449.78	10947.00
	p·none	33032.34	17675.38	12692.18	10275.42
	f·file-LO	7405.04	4009.03	3007.21	2473.82
	p·file-LO	6917.62	3633.05	2564.76	2059.44
	p·icoq-LO	3339.82	1866.04	1370.73	1160.28
	f·file-CI	7577.00	4127.27	3059.70	2536.91
	p·file-CI	7040.39	3781.12	2704.12	2203.67
	p·icoq-CI	3542.08	1990.56	1478.36	1247.65

1.1×, and 1.0×, respectively. These latter modest improvements may be due to restrictions in project file dependency graphs; yet, having two parallel jobs nearly always gives a sizable speedup compared to sequential checking.

RQ2: For Verdi in p·none, going from sequential to 4-way parallel fine-grained checking brings a speedup of 3.2×. The same speedups for Coquelicot, Fin-

map, Flocq, Fomegac, and Surface Effects are 2.0×, 1.8×, 2.0×, 1.5×, and 2.2×, respectively, indicating greater potential for improvements per core than `f•none`. Speedups compared to 4-way parallelism via `f•none` are noteworthy for some projects, e.g., Fomegac (1.4×) and Surface Effects (2.2×). However, Finmap and Flocq are consistently slower to check with `p•none` than `f•none`; this may be due to both projects having many short-running proofs.

RQ3: For Verdi in `f•file-CI`, going from sequential to 4-way parallel coarse-grained checking brings a speedup of 3.0×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.5×, 1.0×, 1.7×, 1.0×, and 1.0×, respectively. The corresponding speedups in LO-Env are essentially the same for most projects. Compared to `f•none` with 4-way parallelization, the speedup for Verdi in CI-Env is 4.3×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.1×, 2.0×, 2.3×, 1.2×, and 1.0×, respectively. This indicates that `f•file` can be an improvement over `f•none` in CISs.

RQ4: For Verdi in `p•file-CI`, going from sequential to 4-way parallel coarse-grained checking brings a speedup of 3.2×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.8×, 2.3×, 1.9×, 1.5×, and 2.0×, respectively. LO-Env gives essentially similar speedups. Compared to `p•none` with 4-way parallelization, the speedup for Verdi in CI-Env is 4.7×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.5×, 3.7×, 2.5×, 1.3×, and 1.0×, respectively.

RQ5: For Verdi in `p•icoq-CI`, going from sequential (ICoq) to 4-way paral-

lelism gives a speedup of 2.8×. The same speedups for Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects are 1.5×, 1.6×, 1.6×, 1.4×, and 1.8×, respectively. Compared to other modes, this mode gives the lowest proof checking times for nearly all projects, both in LO-Env and CI-Env. Compared to `p·file-CI`, which is arguably the most reasonable comparison, `p·icoq-CI` with 4-way parallelization gives speedups for Verdi, Coquelicot, Finmap, Flocq, Fomegac, and Surface Effects of 1.8×, 1.6×, 1.3×, 1.2×, 1.1×, and 1.6×, respectively. The speedup for Verdi in `p·icoq-CI` with 4-way parallelization compared to sequential `f·none` is notable: 28.6×.

3.7 Discussion

Regression proof mode choices: As shown by our results, different modes have widely varying effects on the proof-checking time for different projects. For example, most projects see small or no improvement when switching from `f·none` to `p·none`, while Verdi shows improvement even for only one job. We believe the efficacy of fine-grained parallel checking is directly related to the frequency of long-running proofs in a project, which is relatively high in Verdi, as indicated, e.g., by the number of proof script lines (around 30k). Long-running proofs may allow for fine-grained parallelism to overcome its inherent overhead from task management.

Another consideration when adopting PICOQ for a Coq project is trust in the toolchain. Even though a developer may trust the Coq proof checker, regression proving tools like PICOQ add several additional layers on top of this

checker to trust. Yet, `f·file` and `p·file`, which essentially only rely on file checksumming and `coqdep` for extracting information from `.v` files, require less trust than `p·icoq`, which performs complex analysis of Coq files via plugins. On the other hand, users may opt for quicker feedback using `p·icoq` while still running `f·file` or `p·file` later for higher assurance.

When choosing between `f·file` and `p·file`, a key factor is the file dependency graph of the project. A low number of files itself implies restrictions, but even with many files, dependencies may force certain restricted orders for compilation of `.vo` files; this is particularly apparent for Surface Effects, but also for Coquelicot. Such projects may opt for `p·file` despite its higher overhead. The choice of mode may also depend on the number of available cores in CI-Env. As shown by the `f·none` results for Coquelicot, there may be a breaking point where no further number of jobs and cores decrease coarse-grained proof checking time (but two jobs/cores generally give a large improvement up from just one). In contrast, `p·none` (and therefore `p·file`) can continue to improve with additional cores where `f·none` cannot, but with diminishing marginal returns. The unusual organization of Verdi definitions and proofs into many different files based on type classes and their instances appears to be highly conducive to parallel compilation of files, in a way similar to how module abstraction (functorization) facilitates separate compilation in functional languages [5, 102].

Support for generic definitions: Sozeau and Tabareau introduced support for *generic* Coq definitions that can be used across universes of types [129].

However, Coq's toolchain for asynchronous proof processing ignores universe constraints, since such constraints must be checked for consistency at the global level [133]. This precludes safe use of the modes using fine-grained parallel proof-checking (Section 3.4) in projects that make heavy use of universe polymorphism. One way to address this problem is to leverage the `coqc` option `-vio2vo`, which produces `.vo` files from argument-passed `.vio` files, by asynchronously building the proofs for all proof tasks (possibly in parallel). Once the `.vo` files are produced, they can be loaded into Coq to evaluate universe consistency globally [133]. This approach enables fine-grained parallel proof checking with file selection, but proof selection remains out of reach.

3.8 Threats to Validity

External: Our results may not generalize to all Coq projects. To mitigate this threat, we selected projects that vary in size, number of files, number of proofs, and proof checking time.

Our experiments are executed on a single hardware platform and may not be reproducible. To mitigate this threat, we ran a subset of experiments on our local machines. Although the absolute numbers are not the same as those reported in Section 3.6, proof checking time ratios among our developed techniques remained the same.

We used up to 24 revisions per project. The results may differ for different windows of revisions or longer histories. We selected the latest re-

visions of each project that could be built with Coq version 8.5 (the version supported by the implementation of ICoq, which PICoq extends). Although several projects have long histories, we faced issues similar to those frequently faced when building projects written in other languages [138], e.g., code that cannot be compiled.

Internal: Our implementation of PICoq, as well as our evaluation infrastructure, may contain bugs. To mitigate this, we tested it on changes to small example projects where the expected regression proving outcome was easy to check manually.

The mode for checking that relies on proof selection (`p·icoq`) is subject to the same limitations as ICoq, e.g., with respect to tactic language dependencies and parameterized modules (functors). To trust the results of `p·icoq`, a user must trust Coq's asynchronous proof-checking toolchain as well as the ICoq technique and its implementation in terms of Coq plugins, extensions to `coqc`, and the ICoq proof-level dependency graph analysis.

Construct: We implemented parallel regression proving with selection only for a single proof assistant. Although our techniques and taxonomy should be applicable to other proof assistants (e.g., Isabelle/HOL [111]), future work should confirm the applicability.

3.9 Summary

We presented a taxonomy comprising both state-of-the-art and novel techniques for parallel, incremental regression proving in large-scale verification projects, and their implementation for the Coq proof assistant in the tool PICOQ. In particular, PICOQ is suitable for use in continuous integration systems running on multi-core hardware to quickly verify a project or find failing proofs. By tracking dependencies, PICOQ can avoid checking unaffected files or proofs as changes are made, and check the affected proofs in parallel.

Our evaluation shows that switching from sequential checking in a CIS environment (commonly using Travis CI) to 4-way proof-level parallelization and proof selection with PICOQ can lead to speedups of up to 28.6×. Compared to previous work on sequential proof checking with proof selection in a CIS environment with PICOQ, 4-way proof-level parallelization using PICOQ yields speedups of up to 2.8×. These results indicate the potential of our techniques and PICOQ to increase the productivity of proof engineers.

Chapter 4

MCoq: Mutation Proving for Analysis of Verification Projects

Mutation analysis, which introduces artificial defects into software systems, is the basis of mutation testing, a technique widely applied to evaluate and enhance the quality of test suites. However, despite the deep analogy between tests and formal proofs, mutation analysis has seldom been considered in the context of deductive verification. We propose mutation proving, a technique for analyzing verification projects that use proof assistants. We implemented our technique for the Coq proof assistant in a tool dubbed MCoq. MCoq applies a set of mutation operators to Coq functions and datatypes, inspired by mutation operators previously proposed for functional programming languages. MCoq then checks proofs of lemmas affected by operator application. To make our technique feasible in practice, we implemented several optimizations in MCoq such as parallel proof checking. We applied MCoq to several medium and large scale Coq projects, and recorded whether proofs passed or failed when applying different mutation operators. We then qualitatively analyzed the failed proofs, and found several examples of weak and incomplete specifications. For our evaluation, we made many improvements to serialization of Coq code and even discovered a notable bug in Coq itself,

all acknowledged by developers. We believe MCoq can be useful both to proof engineers for improving the quality of their verification projects and to researchers for evaluating proof engineering techniques.

4.1 Overview

Mutation analysis introduces small-scale modifications to a software system, with each modified system version called a *mutant*. Mutation analysis is widely applied to software systems to perform *mutation testing* [121], where test suites are evaluated on mutants of a system that represent faults introduced by programmers, or are designed to give rise to fault-like behavior. If a specific mutant induces test failures, the mutant is said to be *killed*. However, if a mutant survives all tests, this may indicate an inadequate test suite or present avenues to improve tests. Mutants of a system can be produced in a variety of ways; a common approach implemented for many programming languages, including functional languages such as Haskell [98], is to apply *mutation operators* at a level near the source code syntax. An operator may intuitively represent a particular flaw that programmers are prone to make, such as getting the sign of an integer wrong.

Formal verification can offer guarantees about program behavior and other properties beyond those of testing. In particular, deductive verification using proof assistants is increasingly used for development of trustworthy large-scale software systems [95,103,151]. Nevertheless, just as test suites may be inadequate, formal specifications can be incomplete or fail to account for

unwanted behavior [62, 152], and thus have been the target of mutation analysis [7, 80]. However, mutation analysis has only seldom been considered for proof assistants [104], and, to the best of our knowledge, never with explicit formal proofs in place of tests.

We propose *mutation proving*, a technique for mutation analysis of verification projects using proof assistants, suitable for evaluating the adequacy of collections of formally proven properties of programs. Our technique adapts and extends mutation operators previously used to mutate functional programs. We implemented our technique for the Coq proof assistant [14] in a tool dubbed MCoq. Given a mutation operator and a Coq project, MCoq applies an instance of the operator to a definition in Coq's Gallina specification language, and then checks all proofs that could be affected by the change.

A serious obstacle to operator-based mutation analysis in proof assistants is the extensibility and flexibility of the syntax used to express functions, datatypes, and properties. In particular, Coq supports defining powerful custom *notations* over existing specifications [41], and Coq's parser can be extended with large grammars at any point in a source file by loading plugins [42]. These facilities are convenient for expressing mathematical concepts, but pose a great challenge for processing of Coq documents. Moreover, definitions of functions and datatypes, analogous to classes and methods in Java-like languages, tend to be highly interspersed with proofs, which are analogous to tests [28]. This precludes a simple mutation technique based on text replacement in source files [81].

We overcome these challenges by leveraging the OCaml-based SerAPI serialization library [63], which is integrated with Coq's parser and internal data structures. We extended Coq and SerAPI to support full serialization of all Coq documents used in large scale projects to *S-expressions* (sexps) [108]. We apply our mutation operators to the sexps we obtain, and then deserialize and proof-check the results. To make mutation proving feasible in practice for large-scale Coq projects, we optimized mCoq in several ways, e.g., to leverage multi-core hardware for fast parallel checking of proofs affected by changes after applying a mutation operator.

To evaluate our technique, we applied mCoq to several open source Coq projects, from medium to large scale. We recorded whether a mutant was live or killed based on proofs passing or failing, and then, by manual inspection, qualitatively analyzed why proofs passed or failed for a subset of mutants, finding several weak and incomplete specifications. For our evaluation, we enhanced SerAPI and fixed several serialization issues, significantly increasing its robustness in processing large Coq projects. We also found a notable bug in Coq related to proof processing when applying mCoq, acknowledged and subsequently fixed by the developers [134]. Our technique and tool can be useful both to proof engineers for directly analyzing their verification projects and to researchers for evaluating proof engineering techniques, analogously to how mutation testing is currently used to evaluate testing techniques for functional programs [33].

We believe mutation proving is orthogonal to, and complements, many

other analysis techniques for proof assistants, such as bounded testing [26], dependency analysis [28, 119], counter-example generation [18, 45], property-based testing [31, 122], and theory exploration [90]. Specifically, these techniques do not consider "alternative worlds", where definitions are different from the present ones [80].

This chapter describes the following contributions:

⋆ **Technique**: We propose *mutation proving* for verification projects using proof assistants. We define a set of mutation operators on definitions of functions and datatypes, inspired by operators defined previously for functional and imperative programming languages.

⋆ **Tool**: We implemented mutation proving in a tool, dubbed MCoq, which supports Coq projects. Our tool brings significant extensions to Coq and the SerAPI library for serialization and deserialization of Coq syntax; these extensions pave the way for other transformations of Coq code.

⋆ **Optimizations**: In order to make mutation proving of large projects feasible in practice, we optimized MCoq in several ways to make it run faster. In particular, we implemented several novel forms of parallel checking of affected proofs.

⋆ **Evaluation**: We performed an empirical study using MCoq on 12 large and medium-sized open source Coq projects. For each project, we recorded the number of generated and killed mutants. We qualitatively analyzed a

subset of the mutants and found several weak and incomplete specifications manifested as live mutants.

⋆ **Impact**: Our work resulted in many improvements and bugfixes to SerAPI and enhanced its robustness when applied to large-scale Coq projects, showing that complex, extensible proof documents can be manipulated in a lightweight way. We made several modifications to Coq itself, and these changes have been accepted by Coq developers.

4.2 Background

This section provides a brief background on the Coq proof assistant, the SerAPI library, and mutation testing.

4.2.1 The Coq Proof Assistant

Coq is a proof assistant based on type theory [14], implemented in the OCaml programming language. The specification language of Coq, Gallina, is a small and purely functional programming language. Proofs about Gallina specifications are typically performed using sequences of expressions (tactic calls) in Coq's proof tactic language, Ltac [48]. Source files processed by Coq are sequences of *vernacular commands*, each of which can contain both Gallina and Ltac expressions. Figure 4.1 shows an example Coq source file which defines a function `update` and two lemmas about the function. The intended meaning of `update` is that it returns a new version of a given function `st` from natural numbers to some type `A`, and this returned function maps `h` to `v` but

```
Require Import Arith.

Definition update A (st : nat → A) (h : nat) (v : A) :=
 fun (n : nat) ⇒ if Nat.eq_dec n h then v else st n.

Lemma update_nop : ∀ A (st : nat → A) y v,
 st y = v → update A st y v y = st y.
Proof.
intros; unfold update; case Nat.eq_dec; subst; auto.
Qed.

Lemma update_diff : ∀ A (st : nat → A) x v y,
 x ≠ y → update A st x v y = st y.
Proof.
intros; unfold update; case Nat.eq_dec; congruence.
Qed.
```

Figure 4.1: Example Coq source file `Update.v`.

otherwise behaves as `st`.

Vernacular syntax is extensible by the user in almost arbitrary ways by (1) defining *notations* inside Coq, e.g., `[]` for the empty list constructor `nil`, and (2) loading plugins in Coq that extend syntax. In particular, the Ltac language and basic decision procedures for proof automation are implemented as a collection of plugins. Since plugins can generally be loaded at any time when interacting with Coq, the permitted syntax can grow dynamically as a vernacular file is processed. Hence, writing a robust stand-alone parser for vernacular is difficult, and will break easily as Coq evolves.

Even though Coq provides a logic of total, terminating functions, Ltac allows nontermination, e.g., of proof search. Hence, modifying a Gallina specification or function may result in infinite loops, in analogy with the frequent

infinite loops that arise in tests during mutation testing [121]. The common mitigating practice in mutation testing is to assign execution time thresholds for test execution. Similarly, we set thresholds to the proof checking time for each mutant.

The `coqc` tool compiles source `.v` files to binary `.vo` files and checks all proofs. Such binary files are then loaded by Coq when processing `Require` commands in `.v` files.

4.2.2 SerAPI and Serialization to S-expressions

SerAPI is an OCaml library and toolchain for machine interaction with Coq [63]. SerAPI has two principal components: (1) an interface for serialization and deserialization of Coq syntax and internal data structures to and from S-expressions (sexps) [108] built on OCaml's PPX metaprogramming facilities [113], and (2) a protocol for building and querying Coq documents that abstracts over vernacular commands. In effect, SerAPI overcomes the problem of robustly parsing vernacular by directly integrating with Coq's parsing toolchain and datatypes. Since the serialization routines are automatically generated from Coq's own definitions using metaprogramming, SerAPI is expected to require only modest maintenance as Coq evolves. Before our work, the principal application of SerAPI was for user interfaces for Coq, e.g., web-based interfaces [65].

When mutating Coq projects, we use the SerAPI sexp-based serialization facilities, avoiding heavyweight OCaml library development. Intuitively,

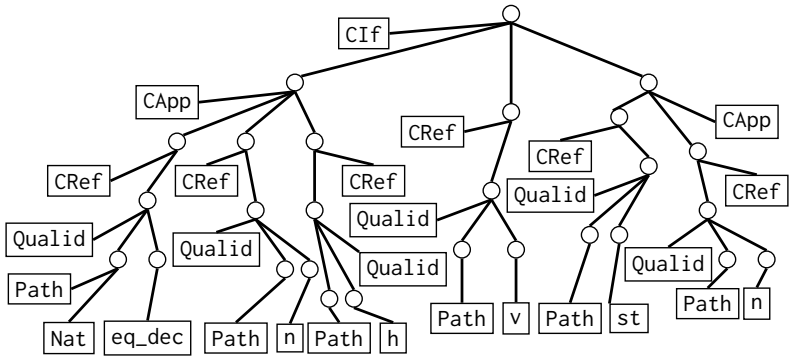


Figure 4.2: Simplified SerAPI sexp of if-expression in Figure 4.1.

a SerAPI sexp is either an atom, representing a constant or variable name, or a list delimited by parentheses. For example, the sexp for the command on the first line in Figure 4.1 is:

```
(VernacExpr()(VernacRequire()(false)(((Qualid(Path)(Arith))))))
```

A more readable but less compact representation of sexps is graphically as trees. For example, the tree of the sexp for the following subexpression is shown in Figure 4.2:

```
if Nat.eq_dec n h then v else st n
```

4.2.3 Mutation Testing and Proving

We follow Papadakis et al. [121] in using *mutation analysis* for the process of generating code variants, and *mutation testing* for the application of this process to support software testing and test suite improvement. In analogy with the latter, we refer to the application of mutation analysis to support proof development using proof assistants and improving collections of formally proven properties as *mutation proving*.

Mutation analysis was proposed by Lipton, then formalized by DeMillo et al. [49], and first applied in practice in the context of software testing by Budd et al. [25]. In mutation testing, test suites that distinguish between a mutant and the original program by reporting an error or violation (i.e., *killing* the mutant), are judged to meet objectives. In contrast, test suites that do not report errors or violations for a mutant (i.e., the mutant is *live*) may not fully meet all objectives and require revision. Intuitively, in these cases, mutants may be viewed as containing buggy code, and the *mutation score* (percentage of killed mutants out of all mutants) as a measure of how well the test suite rules out buggy code.

How to interpret killed and live mutants in mutation proving is less clear than for mutation testing. While there may be definitions of functions or data that are nonsensical for most purposes, a failing proof of a lemma using such definitions does not necessarily indicate an error or mistake (bug) in the definitions; the property may still be true. Coq proof scripts are often brittle [35] and fail to produce proofs when associated definitions are changed in trivial ways. In addition, the goal of a proof assistant verification project may be to prove some lemma unrelated to any specific program.

Nevertheless, live mutants may still indicate inadequacy of the *verification harness* [80] to fully meet reasonable objectives. In particular, live mutants can go *far beyond* flagging up completely unused definitions as in dependency analysis [119]: they can pinpoint that certain *fragments* of key definitions vacuously satisfy behavioral specifications [7], e.g., that an ostensi-

bly strong and complete lemma about a function can be proven regardless of what the returned value is for a certain range of inputs to that function. A low mutation score could indicate the presence of such underspecification in a Coq project, which may eventually manifest as bugs in executable systems [62] and lead to lower trust in formally verified code.

4.3 Technique

In this section, we describe our mutation approach, mutation operators, and optimizations to mutation proving.

4.3.1 Mutation Approach

Our approach to mutation proving follows the classical approach of defining a set of *mutation operators* (operator for short) which describe classes of changes to a project. Intuitively, an operator describes a common mistake made by a proof engineer. When an operator is applied to a project, it may either leave the project unchanged or generate a mutant. When the mutant has been checked, i.e., all relevant proofs have passed, the mutant is declared *live*. Otherwise, if some proof fails, the mutant is considered *killed*.

We define operators for mutation proving as transformations on sexps. For any given sexp, it must be unambiguous whether the transformation can be successfully applied or not. For example, if the transformation pertains to particular constants, it cannot be applied when those constants are absent from the target. The initial step for applying any operator to a verification project

Table 4.1: List of Mutation Operators.

Category	Name	Description
General	GIB	Reorder branches in if-else expression
	GIC	Reverse the order of the constructors in the definition of an inductive type
	GME	Replace expression in the second match case with the expression from the first match case
Lists	LRH	Replace list with head singleton list
	LRT	Replace list with its tail
	LRE	Replace list with empty list
	LRC	Reorder arguments to the list concatenation operator
	LCF	Replace list concatenation expression with the first argument list
	LCS	Replace list concatenation expression with the second argument list
Numbers	NPM	Replace plus with minus
	NZO	Replace zero with one
	NSZ	Replace successor constructor with zero
	NSA	Replace successor constructor with its argument
Booleans	BFT	Replace `false` with `true`
	BTF	Replace `true` with `false`

is to convert all `.v` source files to lists of sexps. For a specific operator *op*, the steps are then to (1) apply *op* to all lists of sexps until a mutant is generated, (2) check the mutated list of sexps, (3) check all source files that transitively depend on the source file that was (indirectly) mutated. The latter three steps are then repeated until no additional mutants can be generated using *op*.

4.3.2 Mutation Operators

Our inspiration for mutation operators for Coq comes from two sources. Primarily, we were inspired by the operators defined by Le et al. [98] for Haskell. Secondarily, we took inspiration from the operators in mutation frameworks for Java such as PIT [39] and the Major framework [92]. We

```
Require Import List. Import ListNotations.

Fixpoint filterMap {A B} (f : A → option B) l : list B :=
match l with
| [] ⇒ [] | x :: xs ⇒
  match f x with
  | None ⇒ filterMap f xs | Some y ⇒ y :: filterMap f xs
  end
end.

Lemma filterMap_conc : ∀ A B (f : A → option B) xs ys,
 filterMap f (xs ++ ys) = filterMap f xs ++ filterMap f ys.
Proof.
induction xs; intros; simpl in *; auto.
case (f a) eqn:?; simpl; auto using f_equal.
Qed.

Lemma filterMap_in : ∀ A B (f : A → option B) a b xs,
 f a = Some b → In a xs → In b (filterMap f xs).
Proof.
induction xs; simpl; auto.
case (f a0) eqn:?; simpl; intuition (auto; congruence).
Qed.
```

Figure 4.3: Example Coq source file using lists `FilterMap.v`.

considered these operators through the lens of our experience from using Coq for 17 years (cumulative).

Table 4.1 lists our operators; for each operator, we give a category, a short name which we will use in the rest of text, and a short description. The General category includes operators which are applicable regardless of whether a project uses a specific datatype from the Coq standard library. The Lists category includes operators which pertain to the ubiquitous list datatype in the standard library. The Numbers category includes operators which apply to natural numbers in their standard linear-size Peano encoding (e.g., 2 is

defined as the successor constructor applied two times to the zero constructor). Similarly, the Booleans category applies to booleans as defined in the standard Coq library.

In contrast to imperative languages such as Java, where numeric datatypes are typically built-in, Gallina has only a few native constructs, which is reflected in the limited number of operators in the General category. Other operators require a project to use the corresponding notations and constants from the standard library; the associated categories therefore pertain to the most elementary and widely used parts of the library.

To illustrate how our operators work, we give a few examples using the Coq code in Figure 4.1 and Figure 4.3. For a more intuitive presentation, we describe the effect of operators mostly in terms of the source code rather than via sexps.

General mutation example: Applying the operator GIB to the file `Update.v` shown in Figure 4.1 results in one mutant `update` has the expressions `v` and `st n` swapped:

```
fun (n : nat) ⇒ if Nat.eq_dec n h then st n else v
```

The proof of `update_nop` does not fail for the mutant, indicating that the lemma does not express any fundamental property of `update`. However, the proof of `update_diff` fails (specifically, `congruence` fails), killing the mutant. Note that the mutation can be performed at the sexp level by swapping the two rightmost subtrees below `CIf` in Figure 4.2.

Lists mutation examples: The source file in Figure 4.3 defines a recursive function `filterMap` that applies a given partial function `f` to a list. The two accompanying lemmas express some basic properties about the function; in particular, `filterMap_conc` establishes that `filterMap` distributes over list append. Applying the operator LRH results in a mutant where the singleton list `[y]` has replaced `y :: filterMap f xs` in `filterMap`. This mutant is killed by the failure of the proof of `filterMap_conc`, since this property no longer holds. Applying the operator LRT results in a mutant where the (tail) list expression `filterMap f xs` has replaced `y :: filterMap f xs`. This mutant survives `filterMap_conc`, but is killed by `filterMap_in`.

4.3.3 Mutation Optimizations

Mutation analysis is generally acknowledged to be a costly process [77], and this also holds true for mutation proving. In this section, we describe several optimizations to our basic mutation proving approach and workflow from above.

In mutation testing, optimizations are generally about generating faster, smarter, or fewer mutants [78, 144]. We focus on accomplishing *faster* mutation, and the insight we build on is that proof checking for mutation proving can be viewed as a particular instance of *regression proving* (covered in chapters 2 and 3), i.e., to check an existing Coq project after a change has been made; similar insights are found in regression testing [32].

Proof selection: A proof selection technique uses knowledge of *modified* files

(or proofs) in a project to only check *impacted* files (or proofs). Since a successful application of a mutation operator means that a sexp file was modified, we can use change impact analysis to perform selective checking during mutation proving.

Proof checking parallelization: Unlike test execution in Java-like languages, proof checking in proof assistants is deterministic, which increases the potential for parallelization on multi-core hardware. In particular, proof checking using Coq is routinely parallelized at the file level, where the main restriction on the degree of parallelism is the file dependency graph.

Mutation operator parallelization: Since we only perform first-order mutation [89], application of one mutation operator to a project can be performed completely independently of application of another operator. Hence, when the goal is to apply several operators to the same project, the outcomes can be computed in parallel, as in mutation testing of software [78].

Mutant parallelization: Application of one mutant can be performed completely independently of application of another mutant. We thus also introduce a parallel mode where each mutant is checked as a separate task.

4.4 Implementation

In this section, we describe the components of the toolchain that we used to implement MCoq, define and discuss our mutation procedure which uses the components, and outline the impact of our toolchain development on

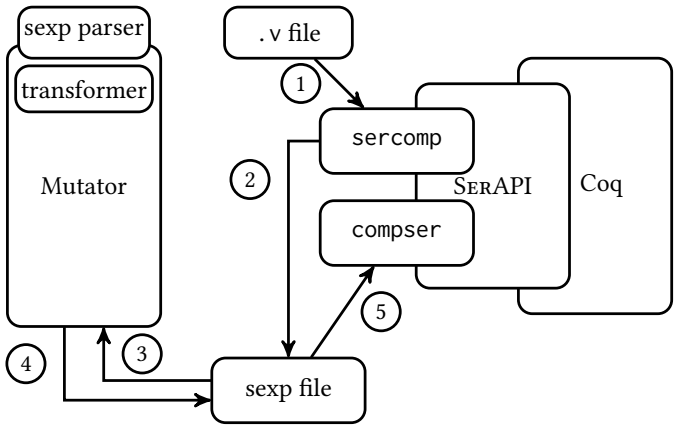


Figure 4.4: mCoq implementation architecture.

other projects.

4.4.1 Toolchain

Our toolchain for mutation proving is implemented in OCaml, Java, and Bash. Figure 4.4 shows an overview of the architecture of the toolchain, and highlights how the main components interact. During mutation proving, Coq source files to be mutated are first given as input to our `sercomp` program integrated with SerAPI ①, which produces corresponding files with lists of sexps ②. The sexps are then handed to our Mutator program ③, which performs parsing and applies the transformations corresponding to a specified mutation operator. Ultimately, Mutator outputs mutated sexps ④ which become input to our `compser` program integrated with SerAPI ⑤. We next describe in more detail each of the main components of the toolchain.

`sercomp`: We implemented a command-line program called `sercomp` on top of

SerAPI which takes a regular Coq `.v` source file as input and outputs the corresponding lists of sexps. `sercomp` is now included as part of SerAPI [64].

`compser`: We implemented a command-line program called `compser` on top of SerAPI, meant to be the inverse of `sercomp`. `compser` takes a file with a list of sexps as input and produces a `.vo` file, or simply checks every sexp. The program is now included as part of SerAPI [64].

Coq fork: We forked the `v8.9` branch of the Coq GitHub repository for Coq version 8.9 and modified it to support proper serialization of all internal data structures via SerAPI. In particular, we added support for serialization of Ltac syntax extensions added by the Ssreflect proof language [75] used in many projects. Our changes have been acknowledged by Coq developers and will be included in Coq 8.10.

SerAPI: We extended SerAPI to support serialization and deserialization of all Coq datatypes required to support large verification projects, and in particular, the Ssreflect proof language. All our changes have been added to the SerAPI codebase.

Mutator: We implemented a library for transformation of sexps produced by `sercomp`, and mutation operators that use this library, in Java. We used an existing library, jsexp [109], to parse and encode sexps. Based on our experience, implementing new operators on top of our library is quick and straightforward. On top of our library, we implemented a program dubbed Mutator that takes sexps and an operator name as input, and produces mutated sexps.

Runner: We implemented a program in Java and Bash that uses the above components to perform mutation on a given Coq project, and then computes mutation scores.

4.4.2 Mutation Modes and Procedure

Based on the mutation approach and optimizations in Section 4.3, we define four basic mutation proving *execution modes*:

Default: A simple mode which checks every `.v` file in a project after a mutant is generated, by compiling the `.v` files to `.vo` files in topological order according to the file dependency graph.

RDeps: An advanced mode which checks only `.v` files affected by a mutation, and caches and reverts to unmodified `.vo` files to avoid generating them twice.

Skip: An advanced mode which checks only `.v` files affected by a mutation, and additionally avoids reverting `.vo` files.

Noleaves: A variant of Default which only checks proofs in, but does not generate `.vo` files for, leaf nodes in the file dependency graph. We added this mode to explore if there were any notable speedups gained by avoiding to write `.vo` files with `compser`.

To realize these modes, we implemented the parameterized mutation procedure CHECKOP shown in Figure 4.5 in our Runner program. In the subprocedures (figures 4.6 and 4.7) called by CHECKOP, there are several auxiliary procedures that behave differently depending on the mode:

```
Require: op – Mutation operator
Require: P – Coq Project
 1: procedure checkOp(op, P)
 2:     vFs ← P.vFiles()
 3:     G ← P.dependencyGraph()
 4:     rG ← G.reverse()
 5:     sVFs ← rG.topologicalSort(vFs)
 6:     v ← ∅
 7:     for vF ∈ sVFs do
 8:         v.add(vF)
 9:         checkOpVFile(G, rG, op, sVFs, v, vF)
10:     end for
11: end procedure
```

Figure 4.5: Pseudocode of parameterized mutation procedure (`checkOp`).

⋆ ***revertFile***: For the Default and Skip modes, the file vF is always reverted. For RDeps, vF is never reverted. For Noleaves, vF is reverted only if it is not a leaf node in rG.

⋆ ***getOtherFiles***: For the Default and Noleaves modes, this procedure returns $rG.topologicalSort(sVFs - v)$, whereas for the RDeps and Skip modes, the procedure instead returns $rG.topologicalSort(rG.closure(\{vF\}) - v)$.

⋆ ***revertOtherFilesBefore***: For all modes except the Skip mode, this procedure does nothing. For the Skip mode, it reverts all files in $G.closure(oVFs) - oVFs - \{vF\}$.

⋆ ***revertOtherFilesAfter***: For all modes except RDeps, it does nothing. For RDeps, it reverts all files in $oVFs + \{vF\}$.

On top of these basic modes, we define four parallel modes, which we believed could lead to significant speedups:

```
Require: G – Dependency Graph
Require: rG – Reverse Dependency Graph
Require: op – Mutation operator
Require: sVFs – Topologically sorted .v files
Require: v – Set of visited .v files
Require: vF – .v file
 1: procedure checkOpVFile(G, rG, op, sVFs, v, vF)
 2:     sF ← sercomp(vF)
 3:     mc ← countMutationLocations(sF, op)
 4:     mi ← 0
 5:     while mi < mc do
 6:         mSF ← mutate(sF, op, mi)
 7:         checkOpSexpFile(G, rG, sVFs, v, vF, mSF)
 8:         mi ← mi + 1
 9:     end while
10:     revertFile(vF)
11: end procedure
```

Figure 4.6: Pseudocode of `checkOpVFile` that is called from Figure 4.5.

ParFile: This mode builds on Skip and parallelizes the for loop in the checkOpSexpFile procedure (lines 8 to 13 in Figure 4.7). Parallelization is at the coarse-grained file level.

ParQuick: Like ParFile, this mode builds on Skip and parallelizes the for loop in the checkOpSexpFile procedure (lines 8 to 13 in Figure 4.7). However, parallelization is at the fine-grained level of proofs [9, 120].

ParMutant: This mode builds on RDeps, and checks each mutant in parallel, i.e., we parallelize the while loop in the checkOpVFile procedure (lines 5 to 9 in Figure 4.6).

6-RDeps: In this mode, we organize the operators into groups of six or less, and run these groups in parallel using the RDeps mode. We limit to six groups to match the number of cores in our evaluation machine.

```
Require: G – Dependency Graph
Require: rG – Reverse Dependency Graph
Require: sVFs – Topologically sorted .v files
Require: v – Set of visited .v files
Require: vF – .v file
Require: mSF – Mutated sexp file
 1: procedure checkOpSexpFile(G, rG, sVFs, v, vF, mSF)
 2:     if compser(mSF) ≠ 0 then
 3:         Global.killed[op] ← Global.killed[op] + 1
 4:         return
 5:     end if
 6:     oVFs ← getOtherFiles(G, rG, sVFs, v, vF)
 7:     revertOtherFilesBefore(vF, oVFs)
 8:     for oF ∈ oVFs do
 9:         if coqc(oF) ≠ 0 then
10:             Global.killed[op] ← Global.killed[op] + 1
11:             break
12:         end if
13:     end for
14:     revertOtherFilesAfter(vF, oVFs)
15: end procedure
```

Figure 4.7: Pseudocode of `checkOpSexpFile` that is called from Figure 4.6.

4.4.3 Impact of Toolchain Development

Work on our implementation toolchain resulted in more than 10 merged code contributions to SerAPI. Specifically, we found over 30 failing test cases that were all fixed. Our serialization enhancements to Coq have been merged and are set to be included in Coq version 8.10. When applying mutation proving to a project (StructTact) during our evaluation, we generated a mutant which we checked with both `coqc` and `compser`; the mutant was killed according to the former but not the latter. The discrepancy was due to a serious bug in Coq related to proof processing, acknowledged and subsequently fixed by the developers [134]. This shows that mutation proving development has

significantly improved general Coq tooling.

4.5 Evaluation

We evaluate MCoq by answering three research questions:

RQ1: What is the number of mutants created for large verification project and what are their mutation scores?

RQ2: What is the cost of mutation testing in terms of the execution time and what are the benefits of optimizations?

RQ3: Why are some mutants (not) killed?

We run all experiments on a 6-core Intel Core i7-8700 CPU @ 3.20GHz machine with 64GB of RAM, running Ubuntu 18.04.1 LTS. We limit the number of parallel processes to be at or below the number of physical CPU cores. We next describe the studied projects, our dependent and independent variables, and our results.

4.5.1 Verification Projects Under Study

Table 4.2 lists the Coq projects used in our evaluation; all are publicly available. For each project, we show the project name, URL, the latest SHA at the time of our experiments, number of `.v` files, total lines of code (LOC), specification LOC, and proof script LOC. All LOCs are computed using the `coqwc` tool, which is bundled with Coq. The last two rows of the table show the average and total values across all projects, if applicable.

We selected the projects based on (1) compatibility with Coq version 8.9, (2) their size and popularity, (3) their extensive use of custom functions and datatypes. In particular, we included several utility libraries, such as Coq-std++ and TLC, which contain many basic functions and specifications needed for program verification and thus have more specification LOC than proof script LOC. In contrast, large formalizations of mathematical theories, such as those for the odd order theorem [74], have many times more proof script LOC than specification LOC, and most of the specifications in these projects are statements of theorems, which we do not mutate. Hence, we did not include any of this latter kind of project. We provide a brief description of each project:

Table 4.2: Projects Used in the Evaluation.

Project	**URL**	**SHA**	**#Files**	**LOC**	**Spec. LOC**	**Proof LOC**
ATBR	github.com/coq-community/atbr	366ac237	42	9705	4123	5567
FCSL-PCM	github.com/imdea-software/fcsl-pcm	b34fce32	12	5747	2939	2851
Flocq	gitlab.inria.fr/flocq/flocq	7ec13200	29	24000	5955	18044
Huffman	github.com/coq-community/huffman	50687911	26	5889	1878	4011
MathComp	github.com/math-comp/math-comp	91fa7b57	89	82323	37520	46040
PrettyParsing	github.com/wilcoxjay/PrettyParsing	189a2625	14	1907	1221	705
Bin. Rat. Numbers	github.com/coq-community/qarith-stern-brocot	7b9cc06d	37	35041	5500	29541
Quicksort Compl.	github.com/coq-contribs/quicksort-complexity	0a6eed8b	36	8809	2617	6202
Stalmarck	github.com/coq-community/stalmarck	6932ed8a	38	11266	3552	7698
Coq-std++	gitlab.mpi-sws.org/iris/stdpp	005887ee	43	13715	6882	6852
StructTact	github.com/uwplse/StructTact	82a85b7e	19	4341	2008	2333
TLC	gitlab.inria.fr/charguer/tlc	4babc16c	49	23494	13217	7802
Avg.	n/a	n/a	36.16	18853.08	7284.33	11470.50
Total	n/a	n/a	434	226237	87412	137646

ATBR: A library that encodes Kleene algebras and their decision procedures using automata, e.g., for checking equality of regular expressions [23].

FCSL-PCM: A library that formalizes the notion of a partial commutative monoid, which is a structure often used for reasoning about pointer-manipulating programs.

Flocq: A library that formalizes floating-point number representations and their arithmetic [21], e.g., as in the IEEE 754 standard.

Huffman: A verified implementation of the Huffman coding algorithm and its theory.

PrettyParsing: A library for implementing verified text-based representations of data. The library is used in the Oeuf verified compiler of Gallina to machine code [110].

Bin. Rat. Numbers: A library formalizing rational numbers and rational arithmetic using the so-called Stern-Brocot encoding [112].

Quicksort Compl. A proof of the average-case complexity of the Quicksort algorithm [140].

Stalmarck: A verified implementation of Stålmarck's proof procedure for propositional logic [106].

Coq-std++: A library of utility functions and lemmas, based heavily on Coq's type class mechanisms. The library is used in the Iris framework for separation logic [91].

StructTact: A library of tactics, utility functions and lemmas, many related to lists, originally extracted from the verified implementation of the Raft distributed consensus algorithm [151].

TLC: A comprehensive library of functions, lemmas, and tactics that aims to be an alternative to the Coq standard library [30].

4.5.2 Variables

Independent variables: We manipulate two independent variables in our experiments: mutation operator and execution mode. For the former, we use the 15 operators defined in Table 4.1. For the latter, we use the 8 execution modes described in Section 4.4.2.

Dependent variables: We compute three dependent variables: mutation score, execution cost, and cost reduction. *Mutation score* provides an estimate for the strength of specifications. This metric is computed as the number of killed mutants over total number of mutants; the computation is either per mutation operator or for all mutants at once. *Execution cost* shows time needed to perform mutation proving; this metric can also be reported per mutation operator or for all mutants at once. *Cost reduction* is a percentage of time saved using various execution modes compared to the time needed to perform mutation proving using the Default mode.

Table 4.3: Total Number of Mutants for each Mutation Operator per Project.

Project	**GIB**	**GIC**	**GME**	**LRH**	**LRT**	**LRE**	**LRC**	**LCF**	**LCS**	**NPM**	**NZO**	**NSZ**	**NSA**	**BFT**	**BTF**	**Total**
ATBR	33	21	74	7	7	7	1	1	1	87	43	19	19	17	18	355
FCSL-PCM	0	8	13	8	8	8	0	0	0	2	5	0	0	35	28	115
Flocq	39	14	93	0	0	0	0	0	0	71	54	2	2	45	62	382
Huffman	0	15	45	72	72	72	15	15	15	19	5	5	5	7	7	369
MathComp	0	10	73	58	58	58	12	12	12	114	385	0	0	136	109	1037
PrettyParsing	30	8	68	17	17	17	28	28	28	13	16	3	3	3	3	282
Bin. Rat. Numbers	2	10	52	0	0	0	0	0	0	203	79	4	4	5	6	365
Quicksort Compl.	12	15	77	104	104	104	49	49	49	27	18	30	30	6	7	681
Stalmarck	0	25	129	101	101	101	3	3	3	42	6	1	1	25	24	565
Coq-std++	12	31	149	68	68	68	13	13	13	23	20	22	22	28	14	564
StructTact	7	3	30	9	9	9	2	2	2	12	5	5	5	2	2	104
TLC	4	36	71	38	38	38	5	5	5	23	38	33	33	20	13	400
Avg.	11.58	16.33	72.83	40.16	40.16	40.16	10.66	10.66	10.66	53.00	56.16	10.33	10.33	27.41	24.41	434.91
Total	139	196	874	482	482	482	128	128	128	636	674	124	124	329	293	5219

Table 4.4: Total Number of Killed Mutants for each Mutation Operator per Project.

Project	**GIB**	**GIC**	**GME**	**LRH**	**LRT**	**LRE**	**LRC**	**LCF**	**LCS**	**NPM**	**NZO**	**NSZ**	**NSA**	**BFT**	**BTF**	**Total**
ATBR	32	15	66	7	7	7	1	1	1	84	40	19	19	17	18	334
FCSL-PCM	0	8	11	8	8	8	0	0	0	2	5	0	0	34	28	112
Flocq	37	14	77	0	0	0	0	0	0	68	54	2	2	37	58	349
Huffman	0	13	45	72	72	72	15	15	15	19	4	5	5	7	7	366
MathComp	0	8	73	58	56	58	11	12	12	113	381	0	0	135	108	1025
PrettyParsing	24	2	62	15	15	15	25	28	28	9	6	2	2	1	1	235
Bin. Rat. Numbers	2	10	47	0	0	0	0	0	0	199	75	4	4	5	6	352
Quicksort Compl.	11	11	73	96	84	104	49	49	49	26	14	29	29	6	7	637
Stalmarck	0	20	124	101	82	101	2	3	3	42	2	1	1	23	20	525
Coq-std++	11	15	139	63	63	64	12	12	12	23	17	22	22	27	13	515
StructTact	7	2	29	9	9	9	2	2	2	12	5	5	5	1	1	100
TLC	4	16	62	38	31	38	5	5	5	12	22	22	17	18	11	306
Avg.	10.66	11.16	67.33	38.91	35.58	39.66	10.16	10.58	10.58	50.75	52.08	9.25	8.83	25.91	23.16	404.66
Total	128	134	808	467	427	476	122	127	127	609	625	111	106	311	278	4856

Table 4.5: Mutation Score per Project.

Project	**GIB**	**GIC**	**GME**	**LRH**	**LRT**	**LRE**	**LRC**	**LCF**	**LCS**	**NPM**	**NZO**	**NSZ**	**NSA**	**BFT**	**BTF**	**Total**
ATBR	96.96	71.42	89.18	100.00	100.00	100.00	100.00	100.00	100.00	96.55	93.02	100.00	100.00	100.00	100.00	94.08
FCSL-PCM	n/a	100.00	84.61	100.00	100.00	100.00	n/a	n/a	n/a	100.00	100.00	n/a	n/a	97.14	100.00	97.39
Flocq	94.87	100.00	82.79	n/a	n/a	n/a	n/a	n/a	n/a	95.77	100.00	100.00	100.00	82.22	93.54	91.36
Huffman	n/a	86.66	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	80.00	100.00	100.00	100.00	100.00	99.18
MathComp	n/a	80.00	100.00	100.00	96.55	100.00	91.66	100.00	100.00	99.12	98.96	n/a	n/a	99.26	99.08	98.84
PrettyParsing	80.00	25.00	91.17	88.23	88.23	88.23	89.28	100.00	100.00	69.23	37.50	66.66	66.66	33.33	33.33	83.33
Bin. Rat. Numbers	100.00	100.00	90.38	n/a	n/a	n/a	n/a	n/a	n/a	98.02	94.93	100.00	100.00	100.00	100.00	96.43
Quicksort Compl.	91.66	73.33	94.80	92.30	80.76	100.00	100.00	100.00	100.00	96.29	77.77	96.66	96.66	100.00	100.00	93.53
Stalmarck	n/a	80.00	96.12	100.00	81.18	100.00	66.66	100.00	100.00	100.00	33.33	100.00	100.00	92.00	83.33	92.92
Coq-std++	91.66	48.38	93.28	92.64	92.64	94.11	92.30	92.30	92.30	100.00	85.00	100.00	100.00	96.42	92.85	91.31
StructTact	100.00	66.66	96.66	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	100.00	50.00	50.00	96.15
TLC	100.00	44.44	87.32	100.00	81.57	100.00	100.00	100.00	100.00	52.17	57.89	66.66	51.51	90.00	84.61	76.50
Avg.	94.39	72.99	92.19	97.31	92.09	98.23	93.32	99.14	99.14	92.26	79.86	92.99	91.48	86.69	86.39	92.58

4.5.3 Results

In this section, we answer the three research questions and highlight interesting findings.

4.5.3.1 The Number of Mutants and Mutation Score

Table 4.3 shows the total number of generated mutants for each pair of project (row) and mutation operator (column). Additionally, the last column shows the total number of mutants per project, and the last two rows show the average and total number of mutants per mutation operator. We can observe that GME generates the most mutants, followed by NZO and NPM. On the other hand, NSZ generates the smallest number of mutants, followed by NSA. This indicates that explicit uses of the natural number successor constructor were few for the projects we used in our evaluation. Following the same format, Table 4.4 shows the number of killed mutants for each pair of project and mutation operator. Table 4.5 shows the mutation score for all pairs of projects and mutation operators; n/a indicates mutation score value that cannot be computed because the number of generated mutants is zero. The last column shows the mutation score for *all* mutants in a given project, which is the metric traditionally reported in mutation testing research. We can see that mutation scores vary from 76.50% (for TLC) to 97.39% (for FCSL-PCM).

It is important to note that mutation scores for verification projects are much higher than traditionally seen in mutation testing research. We expected such high scores for several reasons. First, as mentioned in Section 4.2.3,

many Coq proof scripts are brittle and fail after only trivial changes are made to specifications. Second, even robust proofs tend to be tightly coupled to functions and datatypes, in effect exploring them symbolically rather than relying only on externally observable properties such as outputs. This is what enables proofs to, e.g., establish properties about an infinite number of specific datatype instances, which is impossible for traditional unit tests. Two projects are outliers in terms of mutation score (PrettyParsing and TLC) and we come back to this below.

Table 4.6: Proof Checking and Mutation Time in Seconds for Various Modes.

Project	**Checking**	**Sercomp**	**Default**	**RDeps**	**Skip**	**Noleaves**	**ParFile**	**ParQuick**	**ParMutant**	**6-RDeps**
ATBR	44.97	132.08	2501.19	1925.39	1924.98	2504.64	1448.52	1631.20	617.03	924.46
FCSL-PCM	11.58	21.93	173.52	151.73	151.53	173.92	151.77	151.22	53.29	109.21
Flocq	16.95	37.02	774.89	580.84	581.64	774.74	576.70	579.23	165.11	235.20
Huffman	7.48	11.39	185.49	184.11	183.80	186.05	179.49	206.66	61.49	71.47
MathComp	339.06	589.89	9892.13	8427.10	8419.06	9891.71	6853.53	6736.41	4016.36	3916.54
PrettyParsing	4.24	5.51	272.57	213.41	213.46	273.23	211.13	262.38	65.35	89.08
Bin. Rat. Numbers	25.97	16.81	1108.64	996.39	994.69	1109.32	951.04	932.36	282.42	575.80
Quicksort Compl.	17.34	34.16	1620.77	1096.26	1098.21	1622.30	940.70	954.85	365.03	549.23
Stalmarck	8.94	16.42	801.06	507.78	508.93	806.31	475.71	576.70	194.74	229.51
Coq-std++	30.58	56.71	3180.26	2582.84	2582.70	3182.92	2190.74	2396.88	779.02	1127.56
StructTact	3.25	7.18	54.65	40.44	40.73	54.77	39.24	39.47	18.56	18.57
TLC	21.49	44.60	3123.61	1758.27	1758.76	3123.66	1487.46	1559.28	526.30	689.44
Avg.	44.32	81.14	1974.06	1538.71	1538.20	1975.29	1292.16	1335.55	595.39	711.33
Total	531.85	973.70	23688.78	18464.56	18458.49	23703.57	15506.03	16026.64	7144.70	8536.07

4.5.3.2 Performance

Table 4.6 shows the proof checking and mutation proving time for various execution modes. Specifically, the second column shows time to check the project by running the default build commands (`coqc` via `make`) for each project. The third column shows time to process all files in a project with `sercomp`. Recall that we mutate a file by first obtaining the corresponding sexps via `sercomp`, produce a mutant, and then use `compser` to write a `.vo` file back to disk. Clearly, it would be costly to use both `sercomp` and `compser` to proof check all the files in any given project, so we use this combination only on the file being mutated. The fourth column shows time to perform mutation proving using the Default mode. The remaining columns show execution time for mutation proving for various optimization modes.

Due to performing unnecessary proof processing, the Default and Noleaves modes are consistently the slowest, typically by a wide margin. Reasonably, RDeps and Skip give consistent speedups, sometimes substantial, over the basic modes (on average 23% over Default). Nevertheless, some projects such as Huffman show only marginal improvement.

We expected parallel modes to perform better than the advanced sequential modes. However, ParFile and ParQuick were only substantially faster than Skip for some large projects, such as MathComp. This may be due to many mutants being killed quickly before realizing the benefits of parallel checking. For nearly all projects, ParMutant is a clear winner over 6-RDeps and others, its average speedup over Default is 70%.

4.5.3.3 Qualitative Analysis

To qualitatively analyze why mutants are killed or live, we sampled at least two killed mutants and two live mutants from each project and manually inspected them. In particular, we inspected and labeled 35 live mutants with precisely one of the following labels:

⋆ **UnderspecifiedDef**: The live mutant pinpoints a definition which lacks lemmas for certain cases (18 mutants).

⋆ **DanglingDef**: The live mutant pinpoints a definition that has no associated lemma (13 mutants).

⋆ **SemanticallyEq**: The live mutant is semantically equivalent to the original project (4 mutants).

We highlight some interesting representative live mutants labeled with UnderspecifiedDef, and then discuss our experience from the analysis.

GIB mutant in Flocq: This mutant changed a function for arithmetic on binary IEEE 754 floating-point numbers by swapping the branches of the `if-else` expression:

```
Definition Bplus op_nan m x y := match x, y with (* .. omitted .. *)
| B754_infinity sx, B754_infinity sy ⇒
  if Bool.eqb sx sy then x
  else build_nan (plus_nan x y)
```

The mutant reveals that a particular case of binary addition, namely for numbers representing infinities, is not specified by any lemma. Another live GIB

mutant showed the same problem for the analogous definition for subtraction, `Bminus`.

BFT mutant in StructTact: A mutant which changed `false` to `true` in a function named `before_func` on lists highlighted that the function was weakly specified in the library:

```
Fixpoint before_func {A} (f : A → bool) g l :=
 match l with | [] ⇒ False | a :: l' ⇒
 f a = true ∨ (g a = false ∧ before_func f g l')
 end.
```

Further investigation revealed five standalone lemmas about `before_func` in Verdi Raft [151] which had not been factored out to StructTact. Four of these lemmas kill the mutant.

LRT mutant in MathComp: In this mutant, the last empty list `[::]` is removed from an auxiliary function used by an implementation of the merge sort algorithm:

```
Fixpoint merge_sort_push s1 ss := match ss with
 | [::] :: ss' | [::] as ss' ⇒ s1 :: ss'
 | s2 :: ss' ⇒
   [::] :: merge_sort_push (merge s1 s2) ss'
 end.
```

In essence, mutation preserves the functional correctness of sorting. However, the complexity of the sort function changes from $O(n \log n)$ to $O(n^2)$. According to the author of the function, Georges Gonthier, "the key but unstated invariant of `ss` is that its ith item has size 2^i if it is not empty, so that `merge_sort_push` only performs perfectly balanced merges." He concluded that "without the `[::]` placeholder the MathComp sort becomes two

element-wise insertion sort."

BFT in Flocq: In this mutant, `false` is changed to `true` in the following function:

```
Definition shr_1 mrs :=
 let '(Build_shr_record m r s) := mrs in
 let s := orb r s in
 match m with (* ... *) | Zneg (xO p) ⇒
  Build_shr_record (Zneg p) false s
```

Although there are several lemmas about `shr_1` below the definition, none of them touch this particular match case. In fact, there are no lemmas at all about `Zneg` (negative integer) cases of `shr_1`. This indicates that `Zneg` cases in `shr_1` are unused elsewhere, and we found that they are actually assumed away implicitly by guards in lemmas.

Discussion: All killed mutants we sampled were killed by a nearby proof (same file). PrettyParsing and TLC have the lowest mutation scores of all projects; 83.33% and 76.50%, respectively. We expected the utility libraries (Coq-std++, TLC, and StructTact) to have relatively low scores, due to the greater number of functions and datatypes than in more focused verification projects. The relatively high score of Coq-std++, despite its size in terms of LOC, may indicate that most definitions are extensively specified. To corroborate this, the main author of Coq-std++ emphasized in personal communication that he consistently proves several lemmas about each new definition added to the library. The relatively low score of PrettyParsing is likely partly due to including functions and datatypes for pretty-printing of trees adapted from other work with no correctness theorems. The relatively low score of

TLC is not surprising given the library's size in terms of LOC, high number of definitions, and ambition to replace the standard library.

4.6 Threats to Validity

External: Our results may not generalize to all Coq projects. To mitigate this threat, we chose popular projects that differ in domain, size, number of proofs, and proof checking time. As our infrastructure builds on Coq 8.9, we could only use projects that work with this Coq version. We report results for a single hardware platform, and results may differ if experiments are run elsewhere. We ran all our experiments on two platforms, but we reported results only for one of them (more modern) due to space limitations. Although absolute numbers differ across platforms, our conclusions remain unchanged. We only analyzed a subset of killed and live mutants in our qualitative study. Our findings could differ if we had inspected a different set or more mutants. We mitigate this threat by systematically sampling mutants for inspection.

Internal: Our implementation of the tool and/or scripts may have bugs. To mitigate this threat, we performed extensive unit testing of our code. We also checked that results were the same across modes and that execution time differences were negligible across several runs. Finally, during our qualitative analysis, we validated the outcome of each mutant we studied.

Construct: Our work targets only Coq. Nevertheless, many mutation operators described in Section 4.3.2, e.g., all operators in the Lists category, are applicable to projects using other proof assistants, e.g., Lean [47] and Is-

abelle/HOL [111]. However, more research is needed to develop full sets of operators and evaluate mutation proving for other proof assistants.

4.7 Discussion

Mutation operator design: We implemented and experimented with a mutation operator for changing the order of cases in a pattern matching expression, inspired by Le et al. [98]. However, mutants generated by this operator were nearly always killed immediately (stillborn), since Coq pattern matching branches tend to be completely unambiguous, and the strong type system does not permit leaving out matching cases. This illustrates the problem of defining general operators in Gallina, as opposed to operators that depend on the standard library, e.g., addition and subtraction for Peano arithmetic.

Scope of mutation: We do not consider mutation of lemma statements or of Ltac proof scripts. The main reason is that we then largely lose the analogy between mutation proving and mutation testing, since mutation of *test code* is not performed in the latter. Inductive predicates, which are a special form of inductive datatypes, is arguably a borderline case, but we included them for mutation based on their established interpretation as cut-free higher-order Prolog programs [14].

4.8 Summary

We proposed mutation proving, a technique for analyzing verification projects that use proof assistants. We implemented our technique for the Coq proof assistant in a tool dubbed MCoq. MCoq applies a set of mutation operators to Coq definitions of functions and datatypes, inspired by our experience and operators previously defined for functional programming languages. MCoq then checks proofs of lemmas affected by operator application. To make our technique feasible in practice, we implemented several optimizations in MCoq such as parallel proof checking. We applied MCoq to 12 medium and large scale Coq projects, and recorded whether proofs passed or failed when applying different mutation operators. We then qualitatively analyzed the failed proofs, finding several examples of weak specifications. Moreover, our work has already had significant impact on Coq tooling, and our tool helped to uncover a bug in Coq itself. We believe MCoq can be useful both to proof engineers for improving the quality of their verification projects and to researchers for evaluating proof engineering techniques.

Chapter 5

Related Work

This chapter presents an overview of the work related to the contributions of this dissertation.

5.1 Incremental Verification

Kurshan et al. [96] consider the problem of incremental verification of models of systems, assuming full verification is expensive. They suggest techniques based on hashes of *reduced* models to avoid performing re-verification when the required properties still hold in a changed model. This is similar to *smart hashing* in regression testing [69]. Henzinger et al. [84] consider incremental verification of safety properties of programs using model checking. In contrast to regression proving, whose aim is to find failing proofs quickly, their approach uses previous results to attempt to automatically overcome instances where a program change makes verification fail. Bohme et al. [19] introduced partition-based regression verification that partitions the input space and gradually performs verification. Godlin and Strichman [71] define *regression verification* as establishing the equivalence of successive, related versions of programs. In effect, regression verification is a strengthening of regression

testing, which can only provide limited evidence of preserved functionality.

5.2 Parallel and Asynchronous Proof Checking

Coq's 1970s precursor LCF was based on synchronous, sequential interaction between a human prover and the proof tool [148]. This legacy is reflected in Coq's read-eval-print loop, and by extension, in the top-down interaction with Coq files in classic interfaces such as Emacs with Proof General. Over time, both the assumption on synchrony and on sequential interaction have been reconsidered, which enabled us to develop ıCoq.

Support for parallelism in construction and checking of proofs to exploit multi-core hardware has been addressed previously in several proof assistants, notably Isabelle [147] and ACL2 [67, 124]. Isabelle leverages the support for threads in its "host" compiler, Poly/ML, to spawn proof checking tasks processed by parallel workers. Using a notion of *proof promises*, proofs that require some previous unfinished result can proceed normally and become finalized when extant tasks terminate. Isabelle also includes a build system with integrated support for checking of proofs and management of parallel workers. ACL2 uses the thread-based parallelism in LISP systems to, e.g., perform parallel proof discovery and fine-grained proof case checking. The lack of native threads in Coq's host language, OCaml, prevents similar low-cost fine-grained parallelism [147]. However, more coarse-grained parallelism is possible at the level of processes.

Parallelism at the task level usually necessitates support for some form

of asynchrony, which can then also be exploited at the user interface level to provide greater interactivity. Architectural changes in Isabelle towards a document-oriented asynchronous interaction model were pioneered by Wenzel [148], resulting in the Prover IDE (PIDE) framework. PIDE defines an XML-based protocol between a proof assistant backend and clients such as IDEs. Efforts to bring asynchronous interaction to Coq were initiated by Wenzel [146] and Barras et al. [8], resulting in a new Isabelle-inspired document-oriented interaction model and support for asynchronous proof processing in Coq 8.5 [9]. The potential of Coq's new document model to improve user productivity was highlighted in an extension to the Eclipse IDE called Coqoon by Faithfull et al. [58], which performs fine-grained monitoring of changes to Coq files and reactively processes modified definitions and proofs.

5.3 Regression Testing

There has been more than three decades of work on regression testing techniques [117, 154]. These techniques were the *key inspiration* for the work presented in this dissertation. Specifically, our work is closely related to regression test selection (RTS) [16, 54, 69, 100, 117, 118, 125, 127, 136, 154]. Most of the pioneering work on RTS has studied techniques that collect, for each test, fine-grained dependencies, e.g,. statements and methods. These techniques are frequently unsafe (i.e., they may miss to select some affected tests) for modern programming languages. Recently, Gligoric et al. [69] introduced Ekstazi, an RTS technique that collects dynamic file dependencies; Ekstazi is

more inclusive than prior techniques. Interestingly, we have decided to use fine-grained dependencies for proof selection in ICoq. Our insight is that Gallina does not include the language features that make many RTS techniques unsafe for imperative languages, e.g., dependency injection, class inheritance, and macros. To the best of our knowledge, ICoq is the first proof selection tool.

5.4 Build Systems

Our dependency graph in ICoq is similar to dependency graphs seen in build systems like Google's Bazel [10] and Microsoft's CloudMake [37, 57]. Bazel keeps track of dependencies on a level of targets. Similarly to how ICoq discovers changed proofs and definitions, these modern build systems discover affected targets by computing checksums of the files used by the target and then marking all nodes/targets that depend on the modified node/target.

5.5 Mutation Testing of Functional Programs

Le et al. [98, 99] implemented a mutation testing framework, called MuCheck, for the Haskell language. MuCheck applies mutation operators nondeterministically at the level of abstract syntax trees. The framework is applied in a case study involving sorting of lists via the Quicksort algorithm and random tests using the QuickCheck framework. Cheng et al. [33] used MuCheck to evaluate test coverage for Haskell programs. Taylor and Derrick [135] presented a traditional mutation testing framework for Erlang called

mu2.

Duregård [53] proposed a *black-box* approach to mutation testing of Haskell code, on top of the QuickCheck framework. He defines a mutation score as the probability that a random mutant survives a random test case, and a user can obtain an estimate of this probability by sampling. The function under test must be an instance of a specific type class that allows it to be mutated (without modifying it in-place). Braquehais and Runciman [24] presented a Haskell framework, FitSpec, that uses mutation testing to measure adequacy of sets of properties specified in property-testing frameworks such as QuickCheck and SmallCheck. The aim is to assist in determining both the completeness and minimality of a proposed property set. If either is lacking, output from FitSpec can help refining the property set. FitSpec takes a black-box view of mutations, and uses enumeration to produce mutants.

On one hand, black-box mutation can be applied in a wider context than operator-based mutation, e.g., to functions associated with native code (with inaccessible definitions). On the other hand, black-box mutation sometimes requires defining explicit functions that return mutants. While black-box mutation can be implemented in Coq, we believe the purity and simplicity of Coq's Gallina language make the advantages of black-box mutation modest compared to operator-based mutation. Moreover, we intuitively aim to capture common errors by proof engineers, which is more difficult to do in a black-box setting.

5.6 Mutation of Specifications

In MCoq, we took inspiration from Groce et al. [80], who use a mutation analysis approach to improve the process of verification based on model checking. The goal of mutating a model is to facilitate better understanding of whether successful verification actually implies a desired property. In addition to mutating the model, verification is also guided to cover mutated code. Gopinath and Walkingshaw [79] use mutation analysis to evaluate the effectiveness of type annotations in Python programs.

5.7 Analysis and Testing in Proof Assistants

Berghofer and Nipkow first considered random testing to assist users of the proof assistant Isabelle to specify and verify programs [13]. Bulwahn subsequently improved the Isabelle testing facilities [26]. A Coq testing framework was proposed by Paraskevopoulou et al. [122]. Blanchette and Nipkow [18] presented a counterexample generator called Nitpick for Isabelle/HOL. Given a parameterized property (proposed lemma), Nitpick attempts to find concrete instances of the parameters for which the property does not hold. Cruanes and Blanchette later presented a general tool [45] for counterexample generation and showed how to adapt it to dependent type theory, which is the foundation of Coq. Generators can use several backends, e.g., relational model finders and SAT solvers. Johansson [90] proposed a tool for theory exploration in Isabelle/HOL called Hipster, which attempts to prove interesting facts from a given set of definitions.

Focused testing, generation, and exploration can lead to similar conclusions as mutation proving, e.g., that a definition needs to be changed due to being inadequate in some way. However, these techniques do not consider alternative "worlds", where definitions under consideration are different from the present ones, and are thus largely orthogonal to mutation proving. For example, Hipster can be applied to mutants to reveal consequences of alternative definitions, similarly to applying abduction on logical theories.

Chapter 6

Conclusion

One way to develop correct-by-construction software is by using proof assistants, i.e., writing machine-checked proofs of correctness at the level of executable code. Although the obtained guarantees via such development are highly desirable, proof assistants are not currently well adapted to large-scale software development, and are expensive to use in terms of both time and expertise. The contributions of this dissertation address some of the problems.

First, this dissertation introduced regression proof selection, a technique that tracks fine-grained dependencies between Coq definitions, propositions, and proofs, and only checks those proofs affected by changes between two revisions. We instantiated the technique in a tool dubbed ICoq. We applied ICoq to track dependencies across many revisions in several large Coq projects and showed that ICoq substantially outperforms, in terms of proof checking time, both proof checking from scratch and Coq's timestamp-based toolchain for incremental checking. Second, this dissertation described the design and implementation of PICoq, a set of techniques that blend the power of parallel proof checking and proof selection. Our results indicate that proof-level parallelism and proof selection is consistently much faster than both

sequential checking from scratch and sequential checking with proof selection. Third, this dissertation introduced mutation proving, a technique for analyzing quality of verification projects that use proof assistants. We implemented our technique for the Coq proof assistant in a tool dubbed MCoq. MCoq applies a set of mutation operators to Coq functions and datatypes, and then checks proofs of lemmas affected by operator application. We applied MCoq to several medium and large scale Coq projects, and recorded whether proofs passed or failed when applying different mutation operators. We then qualitatively analyzed the failed proofs, and found several examples of weak and incomplete specifications.

Although our work has already had impact on Coq and the Coq community, we believe that we are at the beginning of a new era for proof development where proof engineers benefit from software engineering techniques. We hope to see many new and exciting techniques that improve lives of proof engineers.

Bibliography

[1] Sidney Amani, Alex Hixon, Zilin Chen, Christine Rizkallah, Peter Chubb, Liam O'Connor, Joel Beeren, Yutaka Nagashima, Japheth Lim, Thomas Sewell, et al. Cogent: Verifying high-assurance file system implementations. *Operating Systems Review*, 50(2):175–188, 2016.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He (Jason) Zhang, and Liming Zhu. Large-scale formal verification in practice: A process perspective. In *International Conference on Software Engineering*, pages 1002–1011, 2012.

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

[5] Andrew W. Appel and David B. MacQueen. Separate compilation for Standard ML. In *Conference on Programming Language Design and Implementation*, pages 13–23, 1994.

[6] David Aspinall. Proof General: A generic tool for proof development. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 38–43, 2000.

[7] Thomas Ball and Orna Kupferman. Vacuity in testing. In *Tests and Proofs*, pages 4–17, 2008.

[8] Bruno Barras, Lourdes del Carmen González Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel, and Burkhart Wolff. Pervasive parallelism in highly-trustable interactive theorem proving systems. In *Intelligent Computer Mathematics: MKM, Calculemus, DML, and Systems and Projects*, pages 359–363, 2013.

[9] Bruno Barras, Carst Tankink, and Enrico Tassi. Asynchronous processing of Coq documents: From the kernel up to the user interface. In *International Conference on Interactive Theorem Proving*, pages 51–66, 2015.

[10] Bazel - Blog. `https://bazel.io/blog/`.

[11] Kent Beck. *Extreme Programming Explained: Embrace Change*. 2000.

[12] Jonathan Bell, Gail E. Kaiser, Eric Melski, and Mohan Dattatreya. Efficient dependency detection for safe Java test acceleration. In *International Symposium on Foundations of Software Engineering*, pages 770–781, 2015.

[13] Stefan Berghofer and Tobias Nipkow. Random testing in Isabelle/HOL. In *International Conference on Software Engineering and Formal Methods*, pages 230–239, 2004.

[14] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[15] Karthikeyan Bhargavan, Barry Bond, Antoine Delignat-Lavaud, Cédric Fournet, Chris Hawblitzel, Catalin Hritcu, Samin Ishtiaq, Markulf Kohlweiss, Rustan Leino, Jay Lorch, et al. Everest: Towards a verified, drop-in replacement of HTTPS. In *Summit on Advances in Programming Languages*, pages 1:1–1:12, 2017.

[16] Swarnendu Biswas, Rajib Mall, Manoranjan Satpathy, and Srihari Sukumaran. Regression test selection techniques: A survey. *Informatica (Slovenia)*, 35(3):289–321, 2011.

[17] Dines Bjørner and Klaus Havelund. 40 years of formal methods. In *International Symposium on Formal Methods*, pages 42–61, 2014.

[18] Jasmin Christian Blanchette and Tobias Nipkow. Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In *International Conference on Interactive Theorem Proving*, pages 131–146, 2010.

[19] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. Partition-based regression verification. In *International Conference on Software Engineering*, pages 302–311, 2013.

[20] S. Boldo, J. H. Jourdan, X. Leroy, and G. Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *Symposium on Computer Arithmetic*, pages 107–115, 2013.

[21] S. Boldo and G. Melquiond. Flocq: A unified library for proving floating-point algorithms in Coq. In *Symposium on Computer Arithmetic*, pages 243–252, 2011.

[22] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[23] Thomas Braibant and Damien Pous. Deciding Kleene Algebras in Coq. *Logical Methods in Computer Science*, 8, 2012.

[24] Rudy Braquehais and Colin Runciman. FitSpec: Refining property sets for functional testing. In *International Symposium on Haskell*, pages 1–12, 2016.

[25] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Symposium on Principles of Programming Languages*, pages 220–233, 1980.

[26] Lukas Bulwahn. The new Quickcheck for Isabelle: Random, exhaustive and symbolic testing under one roof. In *Conference on Certified Programs and Proofs*, pages 92–108, 2012.

[27] Jeanderson Candido, Luis Melo, and Marcelo d'Amorim. Test suite parallelization in open-source projects: A study on its usage and impact. In *Automated Software Engineering*, pages 838–848, 2017.

[28] Ahmet Celik, Karl Palmskog, and Milos Gligoric. iCoq: Regression proof selection for large-scale verification projects. In *Automated Software Engineering*, pages 171–182, 2017.

[29] Ahmet Celik, Karl Palmskog, and Milos Gligoric. A regression proof selection tool for Coq. In *International Conference on Software Engineering, Demo*, pages 117–120, 2018.

[30] Arthur Charguéraud. The optimal fixed point combinator. In *International Conference on Interactive Theorem Proving*, pages 195–210, 2010.

[31] 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 *Symposium on Operating Systems Principles*, pages 270–286, 2017.

[32] Linchao Chen and Lingming Zhang. Speeding up mutation testing via regression test selection: An extensive study. In *International Conference on Software Testing, Verification, and Validation*, pages 58–69, 2018.

[33] Yufeng Cheng, Meng Wang, Yingfei Xiong, Dan Hao, and Lu Zhang. Empirical evaluation of test coverage for functional programs. In *Inter-*

national Conference on Software Testing, Verification, and Validation, pages 255–265, 2016.

[34] M. Chilowicz, E. Duris, and G. Roussel. Syntax tree fingerprinting for source code similarity detection. In *International Conference on Program Comprehension*, pages 243–247, 2009.

[35] Adam Chlipala. Ltac anti-patterns, 2019. `http://adam.chlipala.net/cpdt/html/Large.html`.

[36] Interview with professor Adam Chlipala at MIT. `https://www.functionalgeekery.com/episode-101-adam-chlipala`.

[37] Maria Christakis, K. Rustan M. Leino, and Wolfram Schulte. Formalizing and verifying a modern build language. In *International Symposium on Formal Methods*, pages 643–657, 2014.

[38] cloc - counts blank lines, comment lines, and physical lines of source code in many programming languages. `https://github.com/AlDanial/cloc`.

[39] Henry Coles. PIT mutation testing, 2010. `http://pitest.org`.

[40] coq-dpdgraph. `https://github.com/Karmaki/coq-dpdgraph`.

[41] Coq Team. Coq manual: Syntax extensions and interpretation scopes, 2019. `https://coq.inria.fr/doc/user-extensions/syntax-extensions.html`.

[42] Coq Team. Coq manual: Utilities, 2019. https://coq.inria.fr/doc/practical-tools/utilities.html.

[43] Coquelicot Git repository. https://scm.gforge.inria.fr/anonscm/git/coquelicot/coquelicot.git.

[44] Coq manual: Utilities. https://coq.inria.fr/refman/Reference-Manual017.html.

[45] Simon Cruanes and Jasmin Christian Blanchette. Extending Nunchaku to dependent type theory. In *HaTT@IJCAR*, pages 3–12, 2016.

[46] CTLTCTL Git repository. https://github.com/coq-contribs/ctltctl.git.

[47] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388, 2015.

[48] David Delahaye. A tactic language for the system Coq. In *Logic for Programming and Automated Reasoning*, pages 85–95, 2000.

[49] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[50] Coq development team. Coq Package Index, 2018. https://coq.inria.fr/opam/www/.

[51] Coq development team. Coq proof assistant home page, 2018. https://coq.inria.fr/.

[52] MathComp development team. Mathematical components project, 2018. https://math-comp.github.io/math-comp.

[53] Jonas Duregård. *Automating Black-Box Property Based Testing*. PhD thesis, Chalmers University of Technology, 2016.

[54] Emelie Engström, Per Runeson, and Mats Skoglund. A systematic review on regression test selection techniques. *Journal of Information and Software Technology*, 52(1):14–30, 2010.

[55] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic-with proofs, without compromises. In *Simple High-Level Code for Cryptographic Arithmetic-With Proofs, Without Compromises*, pages 73–90, 2019.

[56] Sebastian Erdweg, Moritz Lichter, and Weiel Manuel. A sound and optimal incremental build system with dynamic dependencies. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 89–106, 2015.

[57] Hamed Esfahani, Jonas Fietz, Qi Ke, Alexei Kolomiets, Erica Lan, Erik Mavrinac, Wolfram Schulte, Newton Sanches, and Srikanth Kandula. CloudBuild: Microsoft's distributed and caching build service. In *International Conference on Software Engineering, Software Engineering in Practice*, pages 11–20, 2016.

[58] Alexander Faithfull, Jesper Bengtson, Enrico Tassi, and Carst Tankink. Coqoon: An IDE for interactive proof development in Coq. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 316–331, 2016.

[59] Finmap Git repository. `https://github.com/math-comp/finmap.git`.

[60] Flocq Git repository. `https://scm.gforge.inria.fr/anonscm/git/flocq/flocq.git`.

[61] Fomegac Git repository, 2016. `https://github.com/skeuchel/fomegac.git`.

[62] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *European Conference on Computer Systems*, pages 328–343, 2017.

[63] Emilio Jesús Gallego Arias. SerAPI: Machine-Friendly, Data-Centric Serialization for Coq. Technical report, MINES ParisTech, 2016. `https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408`.

[64] Emilio Jesús Gallego Arias. SerAPI: The Coq Se(xp)rialized Protocol, 2019. `https://github.com/ejgallego/coq-serapi`.

[65] Emilio Jesús Gallego Arias, Benoît Pin, and Pierre Jouvelot. jsCoq: Towards hybrid theorem proving interfaces. In *Workshop on User Interfaces for Theorem Provers*, pages 15–27, 2017.

[66] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging mathematical structures. In *International Conference on Theorem Proving in Higher Order Logics*, pages 327–342, 2009.

[67] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[68] GitHub. `https://github.com`.

[69] Milos Gligoric, Lamyaa Eloussi, and Darko Marinov. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[70] Milos Gligoric, Wolfram Schulte, Chandra Prasad, Danny van Velzen, Iman Narasamdya, and Benjamin Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 599–616, 2014.

[71] Benny Godlin and Ofer Strichman. Regression verification: Proving the equivalence of similar programs. *Journal of Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

[72] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[73] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008.

[74] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179, 2013.

[75] Georges Gonthier and Assia Mahboubi. An introduction to small scale reflection in Coq. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.

[76] Georges Gonthier, Beta Ziliani, Aleksandar Nanevski, and Derek Dreyer. How to make ad hoc proof automation less ad hoc. In *International Conference on Functional Programming*, pages 163–175, 2011.

[77] Rahul Gopinath, Iftekhar Ahmed, Mohammad Amin Alipour, Carlos Jensen, and Alex Groce. Mutation reduction strategies considered harmful. *Transactions on Reliability*, 66(3):854–874, 2017.

[78] Rahul Gopinath, Carlos Jensen, and Alex Groce. Topsy-Turvy: A smarter and faster parallelization of mutation analysis. In *International Conference on Software Engineering, Demo*, pages 740–743, 2016.

[79] Rahul Gopinath and Eric Walkingshaw. How good are your types? Using mutation analysis to evaluate the effectiveness of type annotations. In *International Conference on Software Testing, Verification, and Validation*, pages 122–127, 2017.

[80] Alex Groce, Iftekhar Ahmed, Carlos Jensen, Paul E. McKenney, and Josie Holmes. How verified (or tested) is my code? Falsification-driven verification and testing. *Automated Software Engineering*, 25(4):917–960, 2018.

[81] Alex Groce, Josie Holmes, Darko Marinov, August Shi, and Lingming Zhang. An extensible, regular-expression-based tool for multi-language mutant generation. In *International Conference on Software Engineering, Demo*, pages 25–28, 2018.

[82] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017.

[83] R. G. Hamlet. Testing programs with the aid of a compiler. *Transactions on Software Engineering*, 3:279–290, 1977.

[84] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Marco A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, pages 332–358, 2003.

[85] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Automated Software Engineering*, pages 426–437, 2016.

[86] InfSeqExt Git repository. `https://github.com/DistributedComponents/InfSeqExt.git`.

[87] It's Travis CI's 5th birthday, let's celebrate with numbers! `https://blog.travis-ci.com/2016-02-05-happy-fifth-birthday-travis-ci`.

[88] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *Transactions on Software Engineering*, 37(5):649–678, 2011.

[89] Yun Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *International Working Conference on Source Code Analysis and Manipulation*, pages 249–258, 2008.

[90] Moa Johansson. Automated theory exploration for interactive theorem proving. In *International Conference on Interactive Theorem Proving*, pages 1–11, 2017.

[91] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[92] René Just. The Major mutation framework: Efficient and scalable mutation analysis for Java. In *International Symposium on Software Testing and Analysis*, pages 433–436, 2014.

[93] Daniel Kästner, Ulrich Wünsche, Jörg Barrho, 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. In *Embedded Real Time Software and Systems*, pages 1–9, 2018.

[94] Gerwin Klein. Proof engineering considered essential. In *International Symposium on Formal Methods*, pages 16–21. 2014.

[95] 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 *Symposium on Operating Systems Principles*, pages 207–220, 2009.

[96] Hardin Kurshan, R. H. Hardin, R. P. Kurshan, K. L. Mcmillan, J. A. Reeds, and N. J. A. Sloane. Efficient regression verification. In *International Workshop on Discrete Event Systems*, pages 147–150, 1996.

[97] Colin J. W. Kushneryk and Paul D. Barnett. Parallel test execution, 2010. https://patents.google.com/patent/US20120102462A1/en.

[98] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. MuCheck: An extensible tool for mutation testing of Haskell programs. In *International Symposium on Software Testing and Analysis*, pages 429–432, 2014.

[99] Duc Le, Mohammad Amin Alipour, Rahul Gopinath, and Alex Groce. Mutation testing of functional programming languages. Technical report, Oregon State University, 2014.

[100] Owolabi Legunsen, Farah Hariri, August Shi, Yafeng Lu, Lingming Zhang, and Darko Marinov. An extensive study of static regression test selection in modern software evolution. In *International Symposium on Foundations of Software Engineering*, pages 583–594, 2016.

[101] Xavier Leroy. The formal verification of compilers. https://deepspec.org/event/dsss17/leroy-dsss17.pdf.

[102] Xavier Leroy. Manifest types, modules, and separate compilation. In *Symposium on Principles of Programming Languages*, pages 109–122, 1994.

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

[104] Xavier Leroy. QuickChick Interface, 2018. `https://softwarefoundations.cis.upenn.edu/qc-current/QuickChickInterface.html`.

[105] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: Certified causally consistent distributed key-value stores. In *Symposium on Principles of Programming Languages*, pages 357–370, 2016.

[106] Pierre Letouzey and Laurent Théry. Formalizing Stålmarck's algorithm in Coq. In *Theorem Proving in Higher Order Logics*, pages 388–405, 2000.

[107] Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Toward a verified relational database management system. In *Symposium on Principles of Programming Languages*, pages 237–248, 2010.

[108] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[109] Julian Mendez. jsexp, 2019. `https://github.com/julianmendez/jsexp`.

[110] Eric Mullen, Stuart Pernsteiner, James R. Wilcox, Zachary Tatlock, and Dan Grossman. Oeuf: Minimizing the Coq extraction TCB. In *Conference on Certified Programs and Proofs*, pages 172–185, 2018.

[111] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[112] Milad Niqui and Yves Bertot. QArith: Coq formalisation of lazy rational arithmetic. In *Types for Proofs and Programs*, pages 309–323, 2004.

[113] OCaml Labs. PPX, 2017. `http://ocamllabs.io/doc/ppx.html`.

[114] The formalization of the odd order theorem has been completed September 20th 2012. `https://web.archive.org/web/20161113010414/http://www.msr-inria.fr:80/news/the-formalization-of-the-odd-order-theorem-has-been-completed-the-20-septembre-2012`.

[115] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *USENIX Annual Technical Conference*, pages 305–319, 2014.

[116] OCaml Package Manager, 2018. `https://opam.ocaml.org`.

[117] Alessandro Orso and Gregg Rothermel. Software testing: A research travelogue (2000–2014). In *Future of Software Engineering*, pages 117–132, 2014.

[118] Alessandro Orso, Nanjuan Shi, and Mary Jean Harrold. Scaling regression testing to large software systems. In *International Symposium on Foundations of Software Engineering*, pages 241–251, 2004.

[119] Anne Pacelet and Yves Bertot. coq-dpdgraph, 2019. `https://github.com/Karmaki/coq-dpdgraph`.

[120] Karl Palmskog, Ahmet Celik, and Milos Gligoric. piCoq: Parallel regression proving for large-scale verification projects. In *International Symposium on Software Testing and Analysis*, pages 344–355, 2018.

[121] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: An analysis and survey. *Advances in Computers*, 2018.

[122] Zoe Paraskevopoulou, Cătălin Hritçu, Maxime Dénès, Leonidas Lampropoulos, and Benjamin C. Pierce. Foundational property-based testing. In *International Conference on Interactive Theorem Proving*, pages 325–343, 2015.

[123] Frank Pfenning and Christine Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *International Conference on Mathematical Foundations of Programming Semantics*, pages 209–228, 1990.

[124] David L. Rager, Warren A. Hunt, and Matt Kaufmann. A parallelized theorem prover for a logic with parallel execution. In *International Conference on Interactive Theorem Proving*, pages 435–450, 2013.

[125] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G. Ryder, and Ophelia Chesley. Chianti: A tool for change impact analysis of Java programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 432–448, 2004.

[126] Vitor Rodrigues and Matthew Fluet. Surface effects for deterministic parallelism. In *Symposium on Trends in Functional Programming*, 2015. `ftp://ftp-sop.inria.fr/indes/TFP15/TFP2015_submission_5.pdf`.

[127] Gregg Rothermel and Mary Jean Harrold. Analyzing regression test selection techniques. *Transactions on Software Engineering*, 22(8):529–551, 1996.

[128] Ilya Sergey, James R Wilcox, and Zachary Tatlock. Programming and proving with distributed protocols. *Symposium on Principles of Programming Languages*, 2:28:1–28:30, 2017.

[129] Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In *International Conference on Interactive Theorem Proving*, pages 499–514, 2014.

[130] Speeding up the build. `http://docs.travis-ci.com/user/speeding-up-the-build`.

[131] StructTact Git repository. `https://github.com/uwplse/StructTact.git`.

[132] Surfaceeffects Git repository. `https://github.com/esmifro/SurfaceEffects.git`.

[133] Enrico Tassi. Coq manual: Asynchronous and parallel proof processing. `https://coq.inria.fr/refman/Reference-Manual031.html`.

[134] Enrico Tassi. Coq pull request #9206, 2018. https://github.com/coq/coq/pull/9206.

[135] Ramsay Taylor and John Derrick. mu2: A refactoring-based mutation testing framework for Erlang. In *Testing Software and Systems*, pages 178–193, 2015.

[136] Testing at the speed and scale of Google. http://google-engtools.blogspot.com/2011/06/testing-at-speed-and-scale-of-google.html.

[137] Travis CI. https://travis-ci.org.

[138] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. There and back again: Can you compile that snapshot? *Journal of Software: Evolution and Process*, 29(4), 2017.

[139] UniMath Git repository. https://github.com/UniMath/UniMath.git.

[140] Eelis Van der Weegen and James McKinna. A machine-checked proof of the average-case complexity of Quicksort in Coq. In *Types for Proofs and Programs*, pages 256–271, 2009.

[141] Verdi Git repository. https://github.com/uwplse/verdi.git.

[142] Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015.

[143] WAD home page. `https://github.com/Fingertips/WAD`.

[144] Bo Wang, Yingfei Xiong, Yangqingwei Shi, Lu Zhang, and Dan Hao. Faster mutation analysis via equivalence modulo states. In *International Symposium on Software Testing and Analysis*, pages 295–306, 2017.

[145] WeakUpTo Git repository. `https://github.com/coq-contribs/weak-up-to.git`.

[146] Makarius Wenzel. PIDE as front-end technology for Coq. *CoRR*, abs/1304.6626, 2013.

[147] Makarius Wenzel. Shared-memory multiprocessing for interactive theorem proving. In *International Conference on Interactive Theorem Proving*, pages 418–434, 2013.

[148] Makarius Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In *International Conference on Interactive Theorem Proving*, pages 515–530, 2014.

[149] Makarius Wenzel. Interactive theorem proving from the perspective of Isabelle/Isar. In *All about Proofs, Proofs for All*, volume 55. 2015.

[150] 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 *Conference on Programming Language Design and Implementation*, pages 357–368, 2015.

[151] Doug Woos, James R. Wilcox, Steve Anton, Zachary Tatlock, Michael D. Ernst, and Thomas Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Conference on Certified Programs and Proofs*, pages 154–165, 2016.

[152] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *Conference on Programming Language Design and Implementation*, pages 283–294, 2011.

[153] Katherine Q Ye, Matthew Green, Naphat Sanguansin, Lennart Beringer, Adam Petcher, and Andrew W Appel. Verified correctness and security of mbedTLS HMAC-DRBG. In *Conference on Computer and Communications Security*, pages 2007–2020, 2017.

[154] Shin Yoo and Mark Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.