iCoq: Regression Proof Selection for Large-Scale Verification Projects

Ahmet Celik
University of Texas at Austin
Austin, TX-78712, USA
ahmetcelik@utexas.edu

Karl Palmskog
University of Illinois at Urbana-Champaign
Urbana, IL-61801, USA
palmskog@illinois.edu

Milos Gligoric
University of Texas at Austin
Austin, TX-78712, USA
gligoric@utexas.edu

***Abstract*—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.**

I. Introduction

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 [23]. The former includes the proofs of the four-color theorem [27] and the Feit-Thompson odd order theorem in Coq [28], and a proof of the Kepler conjecture in HOL Light [29]; the latter includes the certified seL4 operating system kernel in Isabelle/HOL [35], and the CompCert C compiler in Coq [38].

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 [23]. 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 [61].

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 [40].

We believe that proper tool support for large-scale *proof engineering* using proof assistants is an important and growing concern [34]. 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 [41] are around 50k lines in total [63] 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 [60] 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 [7].

In this paper, we show that potential gains in productivity from Coq's new interaction model go beyond recent application inside IDEs [21]. 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 [31], [53]. (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 [33].) 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

978-1-5386-2684-9/17 © 2017 IEEE

ASE 2017, Urbana-Champaign, IL, USA
Technical Research

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 [9], 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 [10], [18], [24], [37], [42], [43], [46], [47], [52], [64]. 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 [7].

To evaluate ICOQ, we applied it on the version control 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.

We make 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.

II. 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) [44]. 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 [5], 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 1.

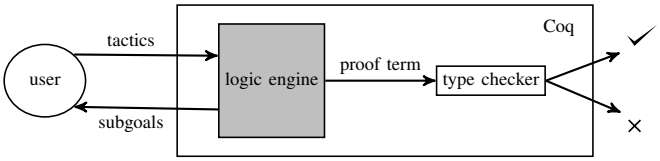


Fig. 1. Coq interactive proof development overview

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 [39]. 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 [16]. `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 [19], [25].

Figure 2 shows 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

Alternate.v

```
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 : forall l, alt [] l l
| alt_step : forall a l t1 t2,
   alt l t1 t2 -> alt (a :: t1) l (a :: t2).

Lemma alt_alternate :
  forall 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.
```

AltLem.v

```
Require Import Alternate.

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

AlternateLem.v

```
Require Import Alternate.

Lemma alternate_alt :
  forall 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 :
  forall l1 l2 l3, alternate l1 l2 = l3 <-> alt l1 l2 l3.
Proof.
intros; split; [apply alternate_alt | apply alt_alternate].
Qed.
```

Fig. 2. Coq Gallina file examples

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 3(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 3(b).

In each Coq file, the commands between `Proof.` and `Qed.` are *proof 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 4. 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 for a list induction principle.

Coq version 8.5, the first stable release to include architectural changes to support a document-oriented interaction model [7], 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 depends on the types (assertions) 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 5 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 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 [51], 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.

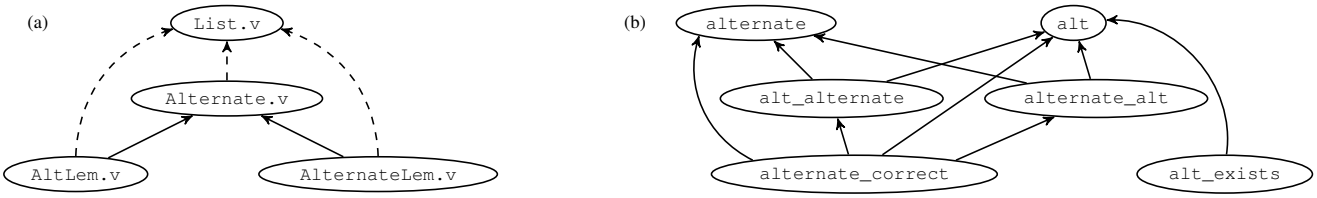


Fig. 3. Coarse- vs. fine-grained dependency graphs for example Coq development

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

Fig. 4. Coq term syntax fragment

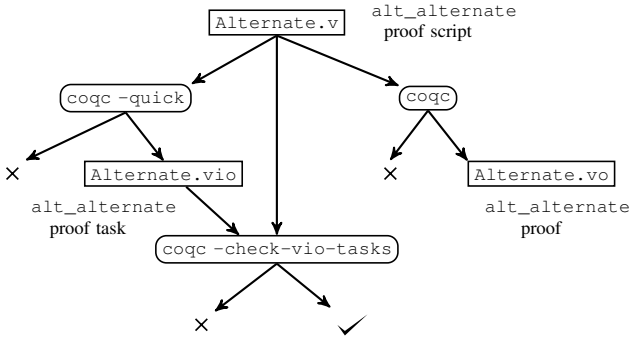


Fig. 5. Coq workflows for `Alternate.v`

III. 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.

A. Phases

Roughly, our technique follows the three phases of a typical regression test selection technique [64]: 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 dependencies 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 [8].

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 Figure 2. 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 3(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 3(b) and add checksums for all nodes. The graphs and checksums are then stored for future use.

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

Fig. 6. Modified Coq function definition in `Alternate.v`

Suppose that the developer of the example Coq project rewrites the definition of the function `alternate` to the one in Figure 6; 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 3(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 3(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 3(b) is up-to-date after the new checksum for `alternate` has been added.

B. 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 8.6 and future Coq versions. Our tools and plugins can also be used (and be useful) outside the context of ICoq.

`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 [2], 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 II, `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. [14]. 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 $\mathcal{C} =$ MD5, is linear in the number of nodes in the tree.

The function V is defined in an obvious way based on the syntax in Figure 4; as an example, Figure 7 shows a fragment of the AST of the proof of `alt_alternate` in `Alternate.v` where V has been applied to each node. 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.

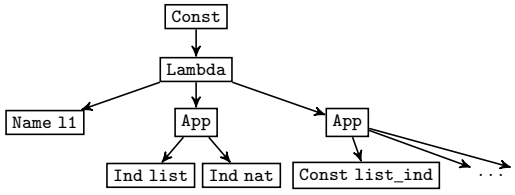


Fig. 7. AST with string values of nodes for example Coq term

`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, `coqdigest` determines that there are two proof tasks

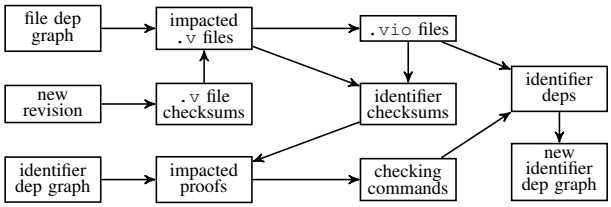


Fig. 8. Toolchain workflow

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, the toolchain workflow (illustrated in Figure 8) follows the general steps of the technique outlined in section III-A. 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, writing the up-to-date graph to disk along with the file-level graph.

TABLE I
VERIFICATION PROJECTS USED IN THE EVALUATION

Project	URL	SHA	LOC	#Revisions
CTLTCTL	[17]	ac57a84f	601	10
InfSeqExt	[32]	5a52a76f	1756	10
StructTact	[50]	8f1bc10a	2496	10
WeakUpTo	[57]	e570e6dc	1819	10
Flocq	[22]	4161c990	24786	24
UniMath	[54]	5e525f08	43049	24
Verdi	[55]	15be6f61	53939	24
$\sum$	N/A	N/A	128446	112
Avg.	N/A	N/A	18349.42	16

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

A. Verification Projects Under Study

Table I shows the list of projects used in our study; all projects are publicly available, all but one on GitHub [3]. 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` [1]), and the number of revisions for the experiments. 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 [62]. 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 [63]. 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 [56]. 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 [13], 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 [12]. We considered revisions of Flocq from Jan to Mar 2016, each consisting of more than 22k library LOC.

B. Variables

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 II (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.

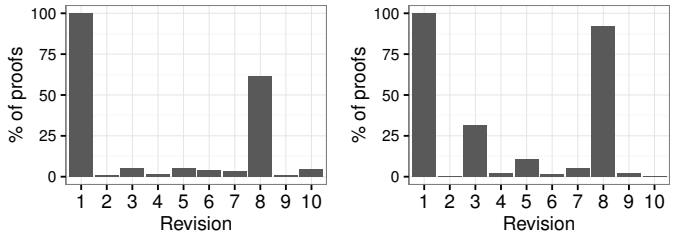
Dependent variables: Our dependent variables measure the effectiveness of proof selection techniques at reducing the amount of effort required to reproof modified programs. To do this we compute the *proof selection percentage* and measure the *proof checking time*. The proof selection percentage is 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 III) of our proof selection technique.

C. Experiment Procedure

Figure 9 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 I), 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 I. 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

```
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
```

Fig. 9. Experiment procedure



(a) InfSeqExt (b) StructTact

Fig. 10. Proof selection percentages for two micro-benchmarks

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 [4], [49], which we can utilize to store the dependencies. Considering that caching is fast and ICoq's dependency files are small, we do not associate 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.

D. Results

We obtained all necessary data by invoking the procedure in Figure 9 twenty-eight times: one invocation for each project in Table I, 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: How effective is ICoq *(compared to the state-of-the-art techniques), i.e., what is the reduction in the number of checked proofs?:* Figure 10 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 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.

TABLE II
TOTAL AND AVERAGE NUMBER OF SELECTED PROOFS AND PROOF CHECKING TIME FOR LARGE VERIFICATION PROJECTS

Project		Proofs			CI-Env Time [s]			LO-Env Time [s]		
		iCoq	Total	P^{sel}	coq_makefile	iCoq	c/i	coq_makefile	iCoq	c/i
Flocq	$\sum$	2164	22482	N/A	888.36	303.71	N/A	297.97	261.62	N/A
	Avg.	90.16	936.75	9.62	37.01	12.65	N/A	12.41	10.90	N/A
UniMath	$\sum$	853	17754	N/A	12882.46	3742.88	N/A	3783.52	1692.33	N/A
	Avg.	35.54	739.75	4.85	536.76	155.95	N/A	157.64	70.51	N/A
Verdi	$\sum$	4458	65413	N/A	32528.57	3379.37	N/A	8157.45	3130.96	N/A
	Avg.	185.75	2725.54	6.80	1355.35	140.80	N/A	339.89	130.45	N/A
	Revision	**iCoq**	**Total**	P^{sel}	**coq_makefile**	**iCoq**	**c/i**	**coq_makefile**	**iCoq**	**c/i**
Verdi (details)	40d0e96f	2748	2748	100.00	1350.26	1375.29	0.98	1355.75	1375.88	0.98
	6b8a7d06	0	2748	0.00	1351.02	63.38	21.31	1.07	5.58	0.19
	56b15cb5	0	2748	0.00	1353.14	3.47	389.84	0.19	4.00	0.04
	9403f6f5	2	2750	0.07	1351.62	148.83	9.08	1347.49	146.63	9.18
	112b39b0	0	2750	0.00	1353.29	3.78	357.44	0.17	4.26	0.04
	57cf9bb1	0	2750	0.00	1352.39	3.72	363.15	0.18	4.47	0.04
	bbf66a54	0	2750	0.00	1349.02	3.71	363.03	0.18	4.43	0.04
	46b6be65	0	2750	0.00	1352.35	3.88	348.36	0.18	4.83	0.03
	27537ec2	0	2750	0.00	1352.00	3.68	366.99	0.19	4.08	0.04
	0f2b8090	0	2750	0.00	1353.03	3.59	376.88	0.17	4.24	0.04
	0201fc23	0	2750	0.00	1353.01	3.62	373.65	0.20	4.43	0.04
	cad0e753	0	2750	0.00	1353.31	3.82	353.43	0.19	4.40	0.04
	2cb92f55	2	2750	0.07	1350.86	147.53	9.15	1346.42	147.20	9.14
	21f660c1	3	2697	0.11	1349.63	64.93	20.78	1351.98	10.11	133.60
	c28a126c	0	2697	0.00	1350.41	3.80	355.09	0.19	6.15	0.03
	57479554	3	2697	0.11	1351.44	64.94	20.81	6.19	8.53	0.72
	ade568dc	0	2697	0.00	1345.22	3.61	372.22	0.20	4.10	0.05
	997ad0a6	0	2697	0.00	1351.46	3.38	399.72	0.21	4.48	0.04
	cee72d1e	3	2697	0.11	1346.28	65.00	20.71	6.14	8.57	0.71
	8ee9b856	0	2697	0.00	1352.78	3.64	371.54	0.17	4.10	0.04
	d4406a1b	0	2697	0.00	1349.74	3.55	379.46	0.20	4.11	0.05
	687a4eaf	1693	2697	62.77	1359.52	1178.12	1.15	1310.31	1169.79	1.12
	06a76847	0	2697	0.00	1383.72	3.51	393.32	0.19	4.06	0.04
	15be6f61	4	2699	0.14	1413.07	216.59	6.52	1429.29	192.53	7.42

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 [24]; 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 a large number of revisions and a low proof selection percentage that is not appropriate to be visualized with a bar chart. Table II shows the results; the table contains two parts, and we discuss each part in turn.

The top part of the table shows result summaries for each project; the sum and the average values are computed across 24 revisions. 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 (CI-Env vs. LO-Env).

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.

TABLE III
RATIO OF TOTAL TIMES FROM TABLE II

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

RQ2: How effective is ICOQ *in terms of the proof checking time in a continuous integration environment?:* 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 II shows the proof checking time. Columns 6 and 7 show the proof checking time for CI-Env when using `coq_makefile` and ICOQ, respectively. Table III 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 with many dependencies and elaborate proofs.

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 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 9 and 10 in Table II show time for `coq_makefile` and ICoq, respectively. 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 III) 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.

V. 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 [47]. 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 [48]. However, Coq's toolchain for asynchronous proof processing ignores universe constraints, since such constraints must be checked for consistency at the global level [51]. 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 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 [24].

VI. 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 version 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 recent studies of regression testing techniques [24].

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.

VII. Related Work

Incremental verification: Kurshan et al. [36] 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 [24]. Henzinger et al. [30] 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. [11] introduced partition-based regression verification that partitions the input space and gradually performs verification. Godlin and Strichman [26] 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.

Parallel and asynchronous proof checking: Coq's 1970s precursor LCF was based on synchronous, sequential interaction between a human prover and the proof tool [60]. 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 Proof General. Over time, both the assumption on synchrony and on sequential interaction have been reconsidered, which enabled us to develop ICoq.

Support for parallelism in construction and checking of proofs to exploit multi-core hardware has been addressed previously in several proof assistants, notably Isabelle [59] and ACL2 [23], [45]. 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 [59]. 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 [60], 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 [58] and Barras et al. [6], resulting in a new Isabelle-inspired document-oriented interaction model and support for asynchronous proof processing in Coq 8.5 [7]. 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. [21], which performs fine-grained monitoring of changes to Coq files and reactively processes modified definitions and proofs.

Regression testing: There has been more than three decades of work on regression testing techniques [42], [64]. These techniques were the *key inspiration* for the work presented in this paper. Specifically, our work is closely related to regression test selection (RTS) [10], [18], [24], [37], [42], [43], [46], [47], [52], [64]. 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. [24] 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.

Build systems: Our dependency graph is similar to dependency graphs seen in build systems like Google's Bazel [8] and Microsoft's CloudMake [15], [20]. 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.

VIII. Conclusions

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.

Acknowledgments

We thank the anonymous reviewers for their comments; Talia Ringer and Zachary Tatlock for their feedback on this work. This work was partially supported by the US National Science Foundation under Grants Nos. CCF-1438982, CCF-1566363, and CCF-1652517, and by a Google Faculty Research Award.

References

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

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

[3] GitHub. https://github.com.

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

[5] D. 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.

[6] B. Barras, L. del Carmen González Huesca, H. Herbelin, Y. Régis-Gianas, E. Tassi, M. Wenzel, and B. 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.

[7] B. Barras, C. Tankink, and E. 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.

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

[9] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[10] S. Biswas, R. Mall, M. Satpathy, and S. Sukumaran. Regression test selection techniques: A survey. *Informatica (Slovenia)*, 35(3):289–321, 2011.

[11] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury. Partition-based regression verification. In *International Conference on Software Engineering*, pages 302–311, 2013.

[12] 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.

[13] 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.

[14] 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.

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

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

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

[18] E. Engström, P. Runeson, and M. Skoglund. A systematic review on regression test selection techniques. *Journal of Information and Software Technology*, 52(1):14–30, 2010.

[19] S. Erdweg, M. Lichter, and W. 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.

[20] H. Esfahani, J. Fietz, Q. Ke, A. Kolomiets, E. Lan, E. Mavrinac, W. Schulte, N. Sanches, and S. Kandula. CloudBuild: Microsoft's distributed and caching build service. In *International Conference on Software Engineering, Software Engineering in Practice*, pages 11–20, 2016.

[21] A. Faithfull, J. Bengtson, E. Tassi, and C. 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.

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

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

[24] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *International Symposium on Software Testing and Analysis*, pages 211–222, 2015.

[25] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. 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.

[26] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Journal of Software Testing, Verification and Reliability*, 23(3):241–258, 2013.

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

[28] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *International Conference on Interactive Theorem Proving*, pages 163–179, 2013.

[29] T. Hales, M. Adams, G. Bauer, T. D. Dang, J. Harrison, L. T. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, Q. T. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, T. H. A. Ta, N. T. Tran, T. D. Trieu, J. Urban, K. Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017.

[30] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. Extreme model checking. In *Verification: Theory and Practice*, pages 332–358, 2003.

[31] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Automated Software Engineering*, pages 426–437, 2016.

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

[33] 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.

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

[35] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Symposium on Operating Systems Principles*, pages 207–220, 2009.

[36] H. 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.

[37] O. Legunsen, F. Hariri, A. Shi, Y. Lu, L. Zhang, and D. 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.

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

[39] G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Toward a verified relational database management system. In *Symposium on Principles of Programming Languages*, pages 237–248, 2010.

[40] The formalization of the odd order theorem has been completed September 20th 2012. http://www.msr-inria.fr/news/the-formalization-of-the-odd-order-theorem-has-been-completed-the-20-septembre-2012.

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

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

[43] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *International Symposium on Foundations of Software Engineering*, pages 241–251, 2004.

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

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

[46] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. 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.

[47] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *Transactions on Software Engineering*, 22(8):529–551, 1996.

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

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

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

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

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

[53] Travis CI. https://travis-ci.org.
[54] UniMath Git repository. https://github.com/UniMath/UniMath.git.
[55] Verdi Git repository. https://github.com/uwplse/verdi.git.
[56] V. Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015.
[57] WeakUpTo Git repository. https://github.com/coq-contribs/weak-up-to.git.
[58] M. Wenzel. PIDE as front-end technology for Coq. *CoRR*, abs/1304.6626, 2013.
[59] M. Wenzel. Shared-memory multiprocessing for interactive theorem proving. In *International Conference on Interactive Theorem Proving*, pages 418–434, 2013.
[60] M. Wenzel. Asynchronous user interaction and tool integration in Isabelle/PIDE. In *International Conference on Interactive Theorem Proving*, pages 515–530, 2014.
[61] M. Wenzel. Interactive theorem proving from the perspective of Isabelle/Isar. In *All about Proofs, Proofs for All*, volume 55 of *Mathematical Logic and Foundations*. 2015.
[62] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Conference on Programming Language Design and Implementation*, pages 357–368, 2015.
[63] D. Woos, J. R. Wilcox, S. Anton, Z. Tatlock, M. D. Ernst, and T. Anderson. Planning for change in a formal verification of the Raft consensus protocol. In *Conference on Certified Programs and Proofs*, pages 154–165, 2016.
[64] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Journal of Software Testing, Verification and Reliability*, 22(2):67–120, 2012.