CoqPIE: An IDE Aimed at Improving Proof Development Productivity

Kenneth Roe(✉) and Scott Smith

The Johns Hopkins University, Baltimore, USA
kendroe@hotmail.com

Abstract. In this paper we present CoqPIE(CoqPIE is available for download at http://github.com/kendroe/CoqPIE), a new development environment for Coq which delivers editing functionality centered around common prover usage workflow not found in existing tools. The main contributions of CoqPIE build from having an integrated parser for both Coq source and for prover output. The primary novelty is not the parser but how it is used: CoqPIE includes tools to carry out complex editing functions such as lemma extraction and replay. In proof replay for example both new and old outputs of the proof script are parsed into ASTs. These ASTs allow replay to do updates such as fixing hypothesis references.

1 Introduction

In this paper we present CoqPIE, a new development environment for Coq which delivers editing functionality centered around common prover usage workflow that is not found in existing tools. The design of CoqPIE was driven by the author's frustrating with a few of the existing proof development workflows. First, when a proof gets to be more than about 300 steps, the time it takes for `coqtop` to process a single tactic slows; this makes browsing quite tedious. Second, when developing a large proof with many lemmas, proving a lemma often reveals an error in the lemma itself. This change then propagates and requires the statements of other lemmas to be changed. Since many of these lemmas have likely already been proven, they need to be replayed (likely with proof script editing), a tedious process.

Improving the above and similar workflows is the primary goal of the design of CoqPIE, which we now describe.

2 An Overview of CoqPIE

The diagram in Fig. 1 shows the CoqPIE UI with a sample proof derivation open. There are three views shown. On the left is a tree view of the entire project similar to the tree view found in modern IDEs. The top level of the tree view shows the files in the project; opening a file node displays a list of all the Coq declarations in that file. Opening a theorem declaration in turn shows

© Springer International Publishing Switzerland 2016
J.C. Blanchette and S. Merz (Eds.): ITP 2016, LNCS 9807, pp. 491–499, 2016.
DOI: 10.1007/978-3-319-43144-4_32

the steps used to prove that theorem, with steps arranged in a tree based on subgoal relationships.

The middle view displays the source file based on the selection made in the tree view on the left. This view functions in a manner similar to the source file view in CoqIDE or Proof General. As with those tools, shading is used to indicate the portion of the file already processed by `coqtop`. Unlike Proof General and CoqIDE, the CoqPIE process management system automatically recompiles dependent source files.

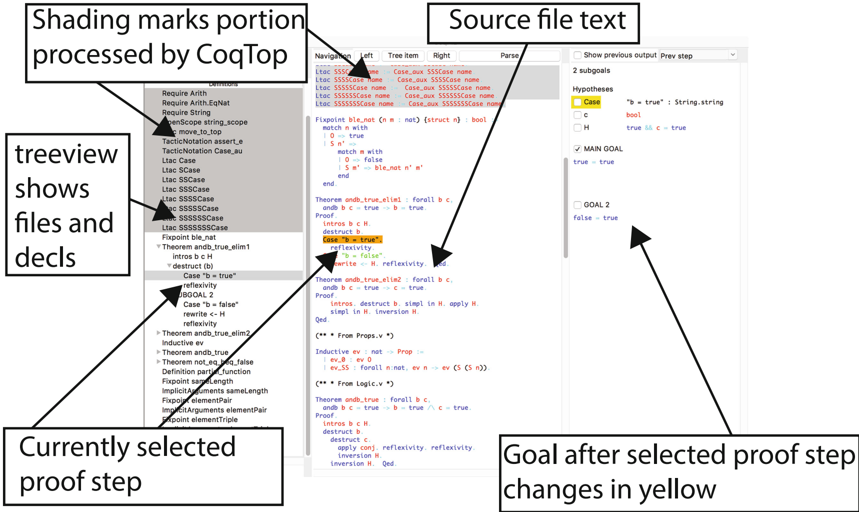

Fig. 1. The main CoqPIE window

The window on the right is similar to the Coq state window in CoqIDE or Proof General: it shows the current goal and hypotheses. However, instead of showing the state at the current processing point of Coq, it shows the state just *after* the selected definition or proof step from the tree view at the left. This is possible because CoqPIE runs the entire project and saves all output from `coqtop` before editing can commence. With this initial pass it is possible to very quickly browse theorems and to see the state after each step. This full proof tree state is also maintained during editing: as the user edits a source file and reruns `coqtop` to verify the updates, the cached outputs are updated. Differences from the state just before the most recent tactic was executed are highlighted in yellow. One can also view differences between hypotheses and the goal or differences between old and new versions of a state (useful for the replay assist described later), via the combo box just above the window on the right which allows selection of which differences to show.

Since CoqPIE keeps intermediate proof state around it can be more intelligent about whether definitions and lemmas are up-to-date: definitions with out-of-date Coq output information are color coded so the user knows they need to be replayed.

Parsing. Coq has an internal CoqAst data structure, but it is not easily accessible with the current API. So, for the current implementation of CoqPIE, we chose to create our own parser. This choice has a number of ramifications. First, the Coq language is quite large and complex; we are only able to parse the commonly-used subset. Second, Coq has a `Notation` construct that can add new syntax to the language. We currently do not have the capability to handle this construct. Longer-term we hope to see a CoqAst API exposed which we will directly be able to use. If a definition or proof step cannot be parsed, then CoqPIE inserts a bad declaration or bad step AST node in the proof tree. The end point is determined by looking for a period.

Dependency management. CoqPIE maintains dependencies between definitions and theorems. When a theorem or definition is changed, all dependent theorems and definitions are highlighted in the project treeview. Creating an exact algorithm for tracking dependencies is very difficult [6] due to the complexities of Coq's higher-order semantics. Many other issues arise in doing dependency analysis, see [22], including opaque vs transparent proof dependencies. An opaque transparency is a dependency that can be identified by the proof statement alone. Transparent dependencies occur when a tactic in the proof script depends on another theorem. These can sometimes be hard to identify as theorems may be chosen automatically by tactics such as `auto`. Our current approach is to use an incomplete dependency tracking algorithm: CoqPIE bases dependency relationships only on identifiers that explicitly appear in a proof or definition.

Lemma extraction. It is often useful to extract one of the goals of a theorem as a lemma in order to break a large proof into more manageable pieces. Coq can process two theorems of 100 steps each much faster than one theorem of 200 steps. CoqPIE provides a command that automates this extraction. The extraction is done in the following steps:

1. The statement of the new theorem is constructed by taking the goal as the consequent. Each hypothesis becomes an antecedent. If the hypothesis appears to be a variable, then it is encoded as part of a `forall` construct. Otherwise it is encoded as an antecedent of the form *hyp* `->`.
2. The steps used to prove the goal are extracted and become the script for the theorem. One can find the end of the sequence of steps used to prove the current goal at the goal state of each subsequent step. The first step after the current step for which the number of goals is one less than that of the current goal is the last step that needs to be extracted with the theorem.
3. In front of the script from the previous step an `intros` statement is added to introduce all of the generated antecedents.

4. The steps to prove the goal are commented out in the main theorem.
5. An `apply` of the newly generated theorem plus an `apply` for each hypothesis is generated in place of those steps that have been commented out.
6. Finally, if there are existential variables in the goal (such as `?508`), the lemma extraction tactic tries to figure out how to fill in this variable. The trick here is to realize that this variable is likely filled in by the steps that prove this goal in the parent theorem. The heuristic is to compare the subgoals after these steps have executed in the main goal to the corresponding subgoals from before they were executed.

This tactic is only a heuristic, and there are several cases in which it will fail. For example, a `Focus` in the middle will break the algorithm for finding the end of the steps for the lemma.

Replay assist. When the statement of a theorem changes, most of the old proof script may still be correct, but at each step minor changes may need to be made. One common example is that hypothesis names may have changed. For example, `apply H` may need to become `apply H0`. To improve the workflow we have implemented a replay assistant which automatically will replay proof and apply heuristics to patch the proof back together. Replay assist saves both the `coqtop` output from before the theorem changed and the output of the new theorem up to the point where a patch may need to be made. One can then compare the two texts and see that `H` has been renamed `H0`, and patch the proof script accordingly.

The replay assistant provides a semi-automated assistant to help with the task of proof patching. There is a "Replay" button that advances `coqtop` past one proof step in a manner similar to "Right." However, steps will be edited if necessary. So, `unfold noFind in H` will be changed to `unfold noFind in H0` if the hypothesis was renamed, and then `coqtop` will advance. There also is a "Show previous output" button to show the old output that can be used to see the old goal state. This is useful if hand editing is necessary. Goal information is attached as annotations to the text of the proof steps. Hence if steps are inserted, then the goals will automatically retain its connection to the original steps.

The current replay algorithm only makes updates to hypothesis labels, but we are planning to extend the functionality in the near future. To update hypothesis labels, CoqPIE finds the renaming by looking at both the old and new result from the previous step and choosing the hypothesis from the new state that is the closest match to the one from the old state. Matches are scored by doing a top down comparison of the two AST trees and counting the number of nodes that match.

Coq users will often explicitly name hypotheses that keep changing position during proof development in order to make direct replay more reliable; while this approach improves the odds of a successful replay, the CoqPIE replay tool allows users to skip this step. In addition, we aim to extend CoqPIE replay to support other changes including detecting when a new subgoal has been added, commenting out a subgoal that has been removed, and reordering proof steps.

Admittedly it will never be possible to patch back every single proof, but it should be possible to eliminate many of the tedious steps users must take when patching a proof.

3 Experience with Implementation

The current implementation has all of the functionality described in this paper. The first author has been using the tool exclusively for proof editing in a multi-file project containing around 10000 lines of Coq code. The tool has also been used to read in a couple of other large derivations including a microprocessor verification example [26][1] and the first few chapters of Software Foundations [20]. We needed to make some very minor edits to get Software Foundations to compile.

There is an up-front cost of using CoqPIE: the full project needs to be run and intermediate goals parsed and cached. The table in Fig. 2 shows times for processing some projects from scratch. The times are taken from runs on a 2011 MacBook Pro with a 2.7 Ghz Intel i5 core and 8G of memory. Since this only needs to re-run if the state of the tool becomes inconsistent, it should be an infrequent event.

Project	Compile time	CoqPIE initialization time	Memory usage (Python process+ largest Coq process)
Model.v	0:03	0:46	35M+163M
DPLL	1:36	9:08	94M+581M
Microprocessor	3:14	4:19:29	12M+825M
Software Foundations	0:06	4:01	47M+187M

Fig. 2. Times and memory usage of CoqPIE on different test cases.

Initialization times for CoqPIE are a few times slower than what is needed to compile the project. While for our current projects the initialization time is tolerable, as shown in the table, for larger projects it will be problematic and we will need to do background updating as is done in PIDE.

Future implementation plans. There are a number of areas where improvement is needed before CoqPIE is ready for widespread adoption. We are looking into integration with PIDEtop. The `coqtop` parser may be integrated directly into CoqPIE if we can get some cooperation from the Coq development team. We plan to add additional heuristics to replay as we work with more complex theorems. We also anticipate adding other high level heuristics beyond replay.

[1] A couple of type checking errors showed up in CoqPIE but not when compiling outside of CoqPIE. We are still working to find the source of these errors.

4 Related Work

In addition to CoqIDE and Proof General, there are several other Coq IDE development efforts. PIDE/jedit [8,27,28] introduces asynchronous communication between the IDE and the theorem prover to improve the user experience. The idea is that as text is being edited in a proof script, the theorem prover is continuously running in the background verifying the new text and all dependencies. Concurrency is used to speed up theorem proving tasks. The tool saves all output and adds markups to the text in appropriate places. Our system currently does not run the prover as a background task or do automatic updating.

CoqPIE provides a goal state window that highlights differences and allows the showing/hiding of individual hypotheses, whereas PIDE/jedit simply stores the text of the theorem prover's output. We do parsing of the output both for the above functionality and replay. CoqPIE also replaces proof scripts with `admit` for proofs on which the user is not working. This gains much of the same performance advantage as concurrency.

The IDE supplied with Coq 8.5 also introduces concurrency and dependency analysis to speed up processing of files. We aim to add support for concurrency in CoqPIE in the future.

Coqoon [15] is an effort to integrate Coq into Eclipse. It provides a tree view to show all files and declarations in the Coq input, similar to our tree view. Parsing is less developed than what exists in CoqPIE: Cocoon provides a simple lexer for tokens and determines the dividing point between definitions by finding periods. CoqPIE on the other hand provides full AST generation along with links between the nodes and positions in the text. There is no concept of storing both the old an new versions of goals in Coqoon and hence no framework for the style of replay assist provided by CoqPIE. Since there are no ASTs, refactoring operations such as lemma extraction are not possible in Coqoon. Finally, there is no difference highlighting since that feature is also dependent on having a full AST. Coqoon is built on top of PIDE and so it allows for asynchronous recompilation of proofs. The PIDE protocol also allows Coqoon to have cached output at each step. The CoqPIE initialization process is not needed; instead, theorem proving is a background task and annotations are collected as they become available.

There also are efforts to build Coq IDEs at MIT and UCSD [3,4]. Both are web-based. However, these tools are primarily intended for teaching.

Proviola [25] is a tool that compiles Coq source code and captures the output at each step. The tool then generates a Javascript-based web page that can display the outputs as the user hovers over each tactic in a proof. Our tool in addition to caching output also parses the output so it can be used by editing macros. CoqPIE also provides algorithms for updating the cache when the source code is edited and the Coq process is rerun.

Pcoq [10] is an earlier UI for Coq. It features a window showing the proof script, another window showing the Coq output and a third window showing a list of potential theorems that can be applied at the current step. The first two windows are similar to what exists in Proof General and Coq IDE. The third

window is unique to Pcoq and would be a useful feature to add to CoqPIE. CtCoq [9,12] builds on Pcoq. It provides the same basic windows as Pcoq, and also parses Coq syntax. It is integrated directly with the CoqAst data structure. Unlike CoqPIE, this AST parsing is used to create a tree-oriented editing paradigm. UI-based point/click/drag and drop commands are used for constructing proofs in place of entering commands. In comparison, our system uses the ASTs to implement many heuristic operations such as replay assist and lemma extraction.

Company Coq [21] is an extension to Proof General that adds many useful features, including shortcut text entry, completion, and reference to Coq documentation. These features would also be useful to add to CoqPIE but they are not our primary focus. Company Coq also includes a lemma extraction feature. However, its implementation does not use an actual AST and hence is less developed.

Proof script transformations have been discussed in [18]. The method involves creating a few correctness preserving transformations. Since the transformations must be formally verified it limits the scope of what tasks can be performed. The refactoring operations in CoqPIE are heuristic in nature so correctness all falls back on Coq.

5 Conclusion

We have presented CoqPIE, a novel Coq editing framework. A key feature of CoqPIE is use of an integrated parser that links AST nodes to source text, which then allows us to create several different forms of intelligent editing functionality, including proof refactoring, showing differences between terms to help guide proof development, and maintaining dependencies so that out-of-date information is clearly highlighted. The current implementation develops a few refactoring tools, but we have only scratched the surface of what refactoring tools can be built over the CoqPIE foundation.

Acknowledgements. The authors would like to thank Gregory Malecha, Valentin Robert and Jesper Bengston for their feedback.

References

1. Coq 8.5 beta release. https://coq.inria.fr/news/123.html. Accessed 20 Mar 2015
2. Coqoon home page. https://itu.dk/research/tomeso/coqoon/. Accessed 19 Mar 2015
3. MIT proofs page. http://proofs.csail.mit.edu/. Accessed 19 Mar 2015
4. Peacoq home page. http://goto.ucsd.edu/peacoq/. Accessed 19 Mar 2015
5. Proof general. http://proofgeneral.inf.ed.ac.uk/. Accessed 20 Mar 2015
6. Alama, J., Mamane, L., Urban, J.: Dependencies in formal mathematics: applications and extraction for Coq and Mizar. In: AISC/MKM/Calculemus, pp. 1–16 (2012)

7. Ayache, N.: Combining the Coq proof assistant with first-order decision procedures (2006)
8. Barras, B., Tankink, C., Tassi, E.: Asynchronous processing of Coq documents: from the kernel up to the user interface. In: Urban, C., Zhang, X. (eds.) ITP 2015. LNCS, vol. 9236, pp. 51–66. Springer International Publishing, Switzerland (2015)
9. Bertot, J., Bertot, Y.: CtCoq: a system presentation. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 231–234. Springer, Heidelberg (1996)
10. Bertot, Y.: Pcoq: a graphical user-interface for Coq. https://www-sop.inria.fr/lemme/pcoq/
11. Bertot, Y.: The CtCoq system: design and architecture. Formal Aspect Comput. **11**(3), 225–243 (1999)
12. Bertot, Y., Kahn, G., Théry, L.: Proof by pointing. In: Hagiya, M., Mitchell, J.C. (eds.) TACS 1994. LNCS, vol. 789, pp. 141–160. Springer, Heidelberg (1994)
13. Boite, O.: Proof reuse with extended inductive types. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 50–65. Springer, Heidelberg (2004)
14. Chlipala, A., Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R.: Effective interactive proofs for higher-order imperative programs. In: 14th ICFP (2009)
15. Faithfull, A., Bengtson, J., Tassi, E., Tankink, C.: Coqoon: an IDE for interactive proof development in Coq. In: Chechik, M., Raskin, J.-F. (eds.) TACAS 2016. LNCS, vol. 9636, pp. 316–331. Springer, Heidelberg (2016). doi:10.1007/978-3-662-49674-9_18
16. Gonthier, G., Mahboubi, A., Tassi, E.: A small scale reflection extension for the Coq system. Research Report RR-6455, Inria Saclay Ile de France (2014)
17. Hasker, R.: The replay of program derivations. Ph.D. thesis, University of Illinois at Urbana-Champaign (1995)
18. Whiteside, I., Aspinall, D., Dixon, L., Grov, G.: Towards formal proof script refactoring. In: Davenport, J.H., Farmer, W.M., Urban, J., Rabe, F. (eds.) MKM 2011 and Calculemus 2011. LNCS, vol. 6824, pp. 260–275. Springer, Heidelberg (2011)
19. Malecha, G., Chlipala, A., Braibant, T.: Compositional computational reflection. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 374–389. Springer, Heidelberg (2014)
20. Pierce, B.C., Casinghino, C., Gaboardi, M., Greenberg, M., Hritcu, C., Sjoberg, V., Yorgey, B.: Software foundations. https://www.cis.upenn.edu/~bcpierce/sf/current/index.html
21. Pit-Claudel, C., Courtieu, P.: Company-Coq: taking proof general one step closer to a real IDE. In: Coq PL (2016)
22. Pons, O., Bertot, Y., Rideau, L.: Notions of dependency in proof assistants. In: UITP (1998)
23. Tankink, C.: PIDE for asynchronous interation with Coq. http://arxiv.org/pdf/1410.8221.pdf
24. Tankink, C.: Proof in context - web editing with rich modeless contextual feedback. In: 10th International Workshop on User Interfaces for Theorem Provers, pp. 42–56 (2012)
25. Tankink, C., Geuvers, H., McKinna, J., Wiedijk, F.: Proviola: a tool for proof re-animation. In: 9th International Conference on Mathematical Knowledge Management (2010)
26. Vijayaraghavan, M., Chlipala, A., Arvind, Dave, N.: Modular deductive verification of multiprocessor hardware designs. In: Kroening, D., Păsăreanu, C.S. (eds.) CAV 2015. LNCS, vol. 9207, pp. 109–127. Springer, Heidelberg (2015)

27. Wenzel, M.: Asynchronous user interaction and tool integration in Isabelle/PIDE. In: Klein, G., Gamboa, R. (eds.) ITP 2014. LNCS, vol. 8558, pp. 515–530. Springer, Heidelberg (2014)
28. Wenzel, M.: Isabelle/jedit (2014). http://isabelle.in.tum.de/dist/doc/jedit.pdf. Accessed 19 Mar 2015