New Emacs Mode For Interaction With Coq Proof Assistant

A Project Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Technology

by

Yuvraj Raghuvanshi
(111801048)

COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY PALAKKAD

CERTIFICATE

*This is to certify that the work contained in the project entitled "**New Emacs Mode For Interaction With Coq Proof Assistant**" is a bonafide work of **Yuvraj Raghuvanshi** (**Roll No. 111801048**), carried out in the Department of Computer Science and Engineering, Indian Institute of Technology Palakkad under my guidance and that it has not been submitted elsewhere for a degree.*

Dr. Piyush P. Kurur

Assistant/Associate Professor

Department of Computer Science & Engineering

Indian Institute of Technology Palakkad

Acknowledgements

I would like to thank Dr. Piyush P. Kurur for floating this project and allowing me to pick it up, giving me both the opportunity to learn a dialect of Lisp, which I consider to be a really elegant family of languages, and also how to customize Emacs, a topic I had been wanting to understand in depth since a year but could not find the time for.

After some experience with Emacs-lisp, I have realized how steep the learning curve is and that it is a very challenging lisp dialect to get right. I would not have been able to continue putting in as much work as learning Emacs lisp development requires had this not been a BTP project, so I'm grateful to the institute for that.

Contents

Chapter 1

Introduction

Currently, the widely preferred way to interact with the Coq proof assistant REPL is to use Emacs with the Proof-General major mode, which provides a generic interface for interacting with various proof assistants, including Coq.

However, following the introduction of Coq STM [1] that came with Coq release 8.5, libraries have been developed that provide a potentially much more powerful interface, due to access to Coq's internal OCaml data types rather than just relying on parsing the output of the Coq REPL.

The goal of this BTP is to make use of the SerAPI library to develop an Emacs major mode for interaction with the Coq proof assistant. The other alternative would be to create a minor mode, which could run alongside Proof General, but that approach has some issues which will be detailed in later sections.

1.1 Coq Proof Assistant

Coq is an interactive formal proof assistant. It provides ways to define mathematical definitions, computable functions or programs, and theorems and allows to write machine-verified proofs for these theorems. It also allows to *extract* executable programs from proofs

of their formal verifications.

The underlying concept behind Coq's machine-verification is that of higher-order type theory- the Calculus of Interactive Constructions wherein propositions are dependent types.

Coq has a dependently typed functional programming language at its core. Not to be confused by an automatic theorem prover, Coq doesn't prove theorems, but verifies the correctness of proofs. However, it does include some limited automated theorem proving *tactics*. So a certifying program (a program that outputs not only the target output but also a proof of correctness of its output) can be used in conjunction with Coq to verify its proof and produce a certified program.

Some popular examples of practical applications of certified programming with Coq include the CompCert project [2] that used Coq to both program a C compiler back-end and prove its soundness, and the machine-verified proof of the four-colour-theorem by Georges Gonthier written using Coq [3].

1.2 Emacs

Emacs is a highly customizable text editor which is more akin to an isolated sandbox system than to a text editor. The base Emacs package comes with support for extending its functionality via packages that are written in Emacs Lisp, a dialect of Lisp.

These extensions usually take the form of what are called Emacs *modes*. There are two types of modes- major and minor. At a given time, an Emacs buffer can only be in one major mode, which defines things such as keymaps and syntax tables that dictate the interaction of the user with Emacs. On the other hand, minor modes are packages that modify or enhance some of the functionalities provided by the major mode a buffer is in. Unlike major modes, several minor modes can be active at any given time in a buffer. The outcome of this BTP is intended to be a major mode.

1.3 Organization of This Report

This chapter provides a background for the topics covered in this report. It provides a brief introduction to the Coq proof assistant and Emacs. The rest of the chapters are organised as follows: the next chapter talks about the existing interaction facility with the Coq REPL in Emacs- Proof General. In Chapter 3, the SerAPI library is introduced. This is the API this BTP will make use of at its core. The chapter will provide details on how SerAPI provides a way to implement IDE features within Emacs. Chapter 4 talks in depth about `sercoq-mode` , which is the major mode this BTP is all about. This chapter goes into some detail about what are the features `sercoq-mode` currently has and how they work. The chapter also talks about the challenges and issues currently present. Finally, the last chapter talks about future development and what phase-2 will involve for `sercoq-mode` .

Chapter 2

Review of Existing Works: Proof General

The interactive interface to Coq proof assistant provided by the implementation is **coqtop**, a top-level REPL, which functions via taking as input proof-building commands one by one, keeping track of all subgoals that are created in the process. In addition to proof-building commands, control commands and commands for requesting information about he underlying structures are also used in the usual theorem proving process.

A command-based REPL model, while good for short and quick proofs, is not the best suited model for larger proofs or complex programs.

Owing to this, **Proof General** [4], which is an Emacs-based interface that provides an abstraction of the Coq REPL at the document-level, is one of the most popular and staple ways to interact with Coq and also other proof assistants like Isabelle, LEGO, and many others.

Proof General provides a convenient document-based model over the underlying command-based REPL. It has been in active development and maintenance for many years and thus provides many useful features to interact with proof assistants.

2.1 Features Provided By Proof General

Proof General provides the following features for Coq. Note that this is not an exhaustive list.

- Syntax highlighting.
- Source code navigation
- Parallel asynchronous compilation
- Project management
- Displaying the current state of goals and hypotheses at all times.
- Working across multiple script files with dependencies.
- Graphical Proof Tree visualisation

2.2 Limitations

Since Proof General works by essentially giving top-level commands to coqtop and parsing its output it has certain limitations that are intrinsic to any such model.

- It doesn't have full access to Coq's internals, which are OCaml datatypes. Until recently, Coq's internals could only be accessed via plugins, which are written in OCaml, and thus support only OCaml-friendly environments.
- Some features that require assistance of the Coq core, such as dynamic tactic extension of user defined tactics are not available with Proof General.

Chapter 3

The SerAPI Library

3.1 Coq STM

Coq 8.5 introduced the State Transactional Machine (STM) and an XML-based interactive protocol. The concept behind the STM API is to represent a Coq script as a directed acyclic graph, with the nodes representing statements (also called Coq sentences), and edges representing dependency of a part of the document on previous parts. The two fundamental operations on this graph are document building and document processing. The former involves adding new nodes to the graph, which is done by parsing a statement successfully, whereas the latter involves processing the already built DAG upto a certain node. Modifying a node has the effect of all its successors (which are dependent on said node) becoming invalid if previously processed.

3.2 SerAPI

SerAPI [5] is a library and protocol for machine-to-machine communication with Coq, built with the main focus of streamlining IDE development for Coq. It provides serialization of Coq's internal datatypes (which are implemented in OCaml) to JSON or sexps, and vice-versa.

This serialization to sexps is incredibly useful in case of Emacs-related development, because Emacs' primary development environment is Emacs Lisp, and Lisp is a homoiconic language, so there is no need for separate parsing libraries as is usually the case with JSON in other languages.

SerAPI is essentially is an evolved version of the STM API, and provides three main sets of operations- document building/checking, querying, and pretty printing. Building is the manipulation of the document DAG as described in the previous paragraph, and querying involves querying Coq for information such as goals, the AST, tactics, definitions, notations, assumptions, lexer tokens, etc.

3.3 Components

SerAPI provides three main components:

- Serlib: SerLib forms the core of SerAPI, providing the serialization of Coq's internal modules and structues to sexps.
- Sercomp: Sercomp is a batch processing and compiler utility for Coq source files.
- Sertop: Sertop is effectively a shell over SerLib, providing a powerful command-based interface for building documents and querying. It has been built with IDE development in mind and therefore is what this BTP will be primarily using.

3.4 Existing Works That Use SerAPI

Several IDE projects have been built using SerAPI, a list containing which can be found on SerAPI's github.

Some notable projects that make use of SerAPI are jsCoq, (which is what motivated the creation of SerAPI), and PeaCoq, which are Coq IDEs that run in a browser environment,

pyCoq, a python interface for SerAPI, and elcoq, an early demo (not functional now) to experiment with the interaction of SerAPI with Emacs.

Chapter 4

Sercoq Mode

The outcome of phase 1 of this BTP is sercoq-mode, a major mode for Emacs that uses the SerAPI Protocol to provide a functional interface for building proofs interactively using Coq.

4.1 Why a major mode?

As mentioned in earlier sections, an emacs mode can be a major or a minor mode. A major mode typically provides a set of interactive functions, a syntax table and a keymap that effectively define how a user interacts with text inside an emacs buffer. Minor modes, on the other hand, aim at providing very specific isolated features which typically don't require the voluntary interaction of the user with the mode's functions. This is why an emacs buffer is always in exactly one major mode at a given time, but it can be in more than one minor modes that support the major mode. For example, when editing an ipython notebook inside emacs, functions like creating/deleting cells, adding figures, running/stopping code execution in one or more cells, etc. define how the user voluntarily interacts with the text (i.e. the source code), so these functions are likely to be part of a major mode, whereas a feature like auto-indentation, syntax-highlighting, autocomplete, etc. works automatically, without requiring constant user invocations, which are functions more suited for a minor

mode. Note that these are all stylistic and design choices, not algorithmic. Determining whether a project warrants a major mode or a minor mode is not an exact science, with sometimes the distinction being significantly less clear than the simple example given here.

There are two main issues that influenced the decision of making a major mode that stands on its own instead of a minor mode that works alongside Proof General. They are the following:

The first is that both Proof General and a SerAPI-based mode attempt to solve the same problem with radically different approaches. Proof General uses the coqtop REPL itself to provide a document-based abstraction over the REPL, whereas SerAPI uses Coq's core building and querying protocols to extract data. Thus, both modes have significant overlap in features they aim to provide but with conflicting approaches, making a strong case for both being separate major modes.

The second issue is that a minor mode implementation that supports and works with Proof General would be coupled with the implementation details and development of Proof General, which is in itself a huge project that has been developed over the course of many years, and would require considerable time to understand the subtleties and internals of.

Keeping in mind these issues, it is fairly clear that a major-mode is the direction to take, and I hope to have convinced the reader as to why.

4.2 Current state of sercoq-mode

The first goal of the project was to get the fundamental and most important features up and running in `sercoq-mode`, so that is what I have worked on in Phase 1 of this BTP.

The mode in its current state allows a user to build proofs and perform computations synchronously while keeping track of the current goals and outputs. Similar to how proof general works, the goals and outputs are shown in their own separate buffers, and are updated with each interaction of the user.

Here is a list of functions that `sercoq-mode` currently provides.

- `sercoq-exec-region` : This interactive elisp function sends the selected region in the emacs buffer to Coq for synchronous processing, updates and displays the goals in the ***sercoq-goals*** buffer, and shows the outputs (if any) in the ***sercoq-responses*** buffer. Simply select a region and run `M-x sercoq-exec-region` to process the region.

  Similar to Proof General, when a text region is processed, `sercoq-mode` locks that region from editing and highlights it to demarcate the processed region from the unprocessed one and so that the user doesn't make accidental changes to the processed region.

- `sercoq-cancel-statements-upto-point` : To edit the a Coq sentence in the processed region, the processing must be reverted back, then the sentence can be edited and processed again. This function does just that. The command `M-x sercoq-cancel-statements-upto-point` "cancels" (cancel is the technical term for undoing a processed statement in SerAPI) all Coq statements that lie between the current point (the position of the cursor in an emacs buffer is called the **point**) and the end of the buffer and makes them editable.

- `sercoq-exec-next-sentence` : Processes the next Coq command in the script. Mapped by default to `C-c C-n` and functions similar to the function in Proof General bound to the same keybind.

- `sercoq-undo-previous-sentence` : Similar to the `C-c C-p` command in Proof General. Undoes the last processed Coq sentence. Mapped to `C-c C-u` by default.

- `sercoq-exec-buffer` : Processes the entire buffer. Mapped to `C-c C-b` by default.

- `sercoq-retract-buffer` : Undoes the processing of everything processed until now. Mapped to `C-c C-r` by default.

- `sercoq-goto-end-of-locked` : Mapped to `C-c C-.` by default, this function moves point (the cursor) to the end of the processed (and therefore locked) region.

The Goal and Response Buffers

The buffers ***sercoq-goals*** shows the current goals and is updated on each command execution. An update can also be manually forced by running the interactive function `sercoq--update-goals` .

The outputs, if any, of the most recently executed Coq sentence are shown in the buffer ***sercoq-response***. However, this doesn't mean the outputs of the previous sentences are lost. A user can hover the mouse pointer over a processed sentence and sercoq mode will display the output the sentence generated, if any.

4.3 An overview of the internals of sercoq-mode

Sercoq mode defines a buffer-local variable (a buffer local variable in emacs is one that has a different copy for each buffer) that maintains the state of the prover in that buffer. The state contains things like the sentence ids of each Coq sentence that has been processed, a mapping from sentence ids to their respective regions, the sertop process associated with the buffer, the region currently being processed (if any), etc.

The sertop process runs asynchronously in the background. Commands are sent to this process and responses are received from the process asynchronously as serialized sexps from sertop that are serialized forms of Coq's corresponding OCaml datatypes. These sexps are parsed and inferred, and the state is updated to achieve the functionality provided by `sercoq-mode` .

There are two broad categories of document processing - Document Building and Execution, and Querying coq for information. The following is an overview of how each of these features are implemented in `sercoq-mode` :

Document Building and Execution

Document building involves parsing Coq sentences, which gives each sentence a unique identifier. As mentioned in the previous sections, a document is essentially represented as a DAG (Directed Acyclic Graph), with each sentence serving as a node. Each sentence also has a parent in the graph, with the first sentence having a default parent with id 1.

Building a document using the SerAPI protocol is done with the Add and Cancel commands. The Add command takes a string (that may contain multiple Coq sentences), parses it, and returns the newly added sentence ids along with information about the locations of those sentences relative to the input string. This location information is used by sercoq mode to map sentence ids to their positions in the buffer, so that certain properties (such as the computation result that appears on hovering the mouse pointer over the sentence) can be added to the text in the buffer. Once a sentence has been parsed and added, it is made read-only in the buffer to prevent modification unless explicitly indicated (by undoing the processed sentence).

Executing sentences is done via the Exec command. It takes a sentence id and executes it, along with unexececuted sentences that it depends on in the DAG representing the document. It is difficult to predict the exact dependency graph of a node, so sercoq mode Execs every newly generated sentence id. This doesn't cause any side effects because Exec'ing the same sentence id twice is a no-op in sertop.

Cancellation of an executed sentence is done using the Cancel command with a list of sentence ids to be cancelled. On successful cancellation, sertop outputs Answer type messages that contain a list of cancelled sentence ids. The canceled sids (sentence ids) are removed from the state, and their corresponding regions are made writable. The goals and response buffers are also updated.

There is a minor caveat with cancellation which is that if an unexecuted sentence is canceled, it forces the execution of all unexecuted sentences before it in the DAG before canceling. This is a hard limitation of the Coq STM and can't be fixed right now, therefore

sercoq mode executes all parsed commands, thus ensuring that any cancellation, if ever done, is only done for executed sentences, avoiding this issue entirely.

Querying

Sertop can be queried for many things such as search, locate, current goals, the AST, notations, definitions, the environment, and several other type of queries.

Out of these, currently, sercoq mode only queries for current goals to update the ***sercoq-goals*** buffer after every execution or cancellation. More query types will be added as the project progresses to leverage the full power of the SerAPI Protocol.

4.4 Challenges and Issues

The main challenge that kept coming up in implementing sercoq mode is that the output from sertop comes asynchronously, and emacs only accepts outputs when it is waiting for user input, or is in a waiting state due to a blocking elisp function call. Due to this, it becomes important to store information about commands that are in flight, i.e., have not been acknowledged yet by sertop in the buffer-local state because their results need not arrive instantaneously.

Another challenge is handling Coq sentences that parse but fail to execute. Take, for example, a recursive function that doesn't have well-formed recursion. The issue is that currently sercoq mode sets a region as read only when a sentence is successfully parsed. This was preferred initially because unlike Exec command, it is easier to check for an error indication in the output of the Add command. However, for proper functioning of the mode, this behaviour will have to be changed to only consider a sentence final when it is successfully executed, not just parsed. This is the top priority fix for the mode right now and is being worked on currently.

Another challenge is with the implementation of the `sercoq-exec-next-sentence` and `sercoq-undo-previous-sentence` functions. How these functions work is by find-

ing and executing/cancelling the next/previous Coq sentence by searching for the next period (.) character that is followed by one or more whitespaces or newlines. Where this method fails is when a comment contains a period followed by whitespaces and newlines, in which case the comment is mistook for a valid Coq sentence, and forces an error. However, `sercoq-exec-region` and `sercoq-cancel-statements-upto-point` work fine with comments and so do the functions that evaluate the entire buffer. A fix for this issue would involve skipping over comments when looking for the next sentences, and will be worked on once the previous issue is fixed.

The final known limitation that sercoq mode currently has is that it only allows the processed region to grow downwards. What this means is that when a region is executed, the region from the beginning of the document until the end of the executed region is locked into read-only mode. Thus, it is not currently possible to say, first execute an arbitrary region near the bottom of the document, then a region near the top. The reason behind this originally was efficiency and reducing design complexity to get a working prototype first, but now that the basic prototyping is done, this limitation will be worked on and fixed as the project progresses.

Chapter 5

Conclusion and Future Work

Once the few issues described in the previous section are fixed, the features of sercoq mode will be extended beyond the basic proof building features that it currently has.

The primary reason of choosing SerAPI to develop an emacs mode is the fact that SerAPI has access to more information about Coq's state during a proof than Proof General and others of the like. Therefore, Phase 2 of the BTP will start with adding more query types. Currently the mode only queries sertop for current goals, but there are about 20 other powerful query types that will provide more advanced functionality.

While features such as code-completion, syntax highlighting, and auto indentation can be implemented using SerAPI, these features aren't prioritised for sercoq-mode because these features can be provided by independent minor modes such as company-coq provides these features for Proof General.

Another reason for not prioritising these features, especially indentation, is that automatic indentation is a notoriously difficult feature to implement correctly, while also not being as important in a language like Coq which doesn't rely on whitespaces as part of its syntax like languages like python does. Thus, these features will only be worked on once and if all other features mentioned are implemented.

5.1 Repository

Sercoq mode can be found hosted on github here: `https://github.com/yuvraj42/sercoq-mode`

References

[1] B. Barras, C. Tankink, and E. Tassi, "Asynchronous processing of coq documents: From the kernel up to the user interface," in *Interactive Theorem Proving*, C. Urban and X. Zhang, Eds. Cham: Springer International Publishing, 2015, pp. 51–66.

[2] X. Leroy, "A formally verified compiler back-end," *J. Autom. Reason.*, vol. 43, no. 4, pp. 363–446, Dec. 2009. [Online]. Available: https://doi.org/10.1007/s10817-009-9155-4

[3] G. Gonthier, "Formal proof-the four-color theorem," 2008.

[4] D. Aspinall, "Proof general: A generic tool for proof development," in *Tools and Algorithms for the Construction and Analysis of Systems*, S. Graf and M. Schwartzbach, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 38–43.

[5] E. J. Gallego Arias, "SerAPI: Machine-Friendly, Data-Centric Serialization for Coq," MINES ParisTech, Tech. Rep., Oct. 2016. [Online]. Available: https://hal-mines-paristech.archives-ouvertes.fr/hal-01384408