Certifying Graph-Manipulating C Programs via Localizations within Data Structures

Overview for Artifact Evaluation, last updated July 24 2019

I: Overview

Option A: a tour via precompiled Docker

We provide a Docker machine that contains a fully functional, compiled, Coq-checked installation of our system. The machine also contains an installation of Emacs with ProofGeneral to allow users to browse our files and "step through" our proofs. To run our Docker machine, proceed as follows:

1. Install Docker from https://www.docker.com/ and start up the Docker daemon on your machine

2. Run `docker pull johndoe2019/ramifycoq`

3. Run `docker run -it johndoe2019/ramifycoq bash`

If you are unfamiliar with Coq and Emacs in a command line setting, please refer to section III on page 3, where we provide a helpful guide. Our work is in `RamifyCoq_VST/RamifyCoq/`. Please see "try it out" below for an example of how a single example is laid out. Next, please see the table on page 2 for an overview of where the key files are located for all our algorithms.

Option B: a GUI-based tour on your own machine

We explain the process of downloading and building our project on your machine. This option takes more time, but the visual aid provided by a GUI will be appreciated by users who wish to step through our proofs line by line. For this you will need a machine with Coq version 8.9.1, which is the latest at the time of writing. We also recommend Emacs + ProofGeneral as an IDE for Coq.

1. Download
   The following link provides a direct download to our project on GitHub:
   https://github.com/anshumanmohan/RamifyCoq_VST/archive/00d9e9d6fb8b3da3beb89e1dd800db5d2dfddc51.zip
   Unzip the project and `cd` to the `RamifyCoq/` directory. The `VST` directory is our chief dependency but we will not interact with it directly.

2. Build (lengthy step, about 60-90 minutes)
   In the RamifyCoq directory, run either `make vstandme7` or `make vstandme3`, depending on whether you would like to dedicate 7 or 3 cores to this task. You will see two warnings at the very top, but these can be safely ignored. The rest of the script will be a series of "`COQC filename`" commands. You will see three ignorable warnings from VST when building the file `veric/mpred.v` and a few ignorable warnings about reused notations.

Try it out!

For a quick taste, let us examine `find` from Fig 1 of the paper. The hyperlinks that follow lead back into the GitHub repository. The mathematical graph (§4) for `find` is built using a generic PreGraph, a suitable LabeledGraph atop the PreGraph, and a suitable GeneralGraph atop that LabeledGraph. The spatial representation (§5) of this graph is built incrementally over several steps to improve code reusability, but it comes together here in our code. Finally, you can explore the C code of union-find, the Coq-readable AST of that code generated using VST's clightgen tool, and the Coq verification of that AST.

This completes our quick "kick-the-tires" overview. We provide more details in the next section.

II: Step by Step Instructions

Just like `find` described above, we have similar developments for each of our examples. As explained in our paper, we enjoy code reuse with the mathematical and spatial graphs, but the C code, the AST, and the verification files are individually customised.

We verified the following algorithms:

	Algorithm	Code File	Verification File
1	Marking a bigraph	mark_bi.c	verif_mark_bi.v
			verif_mark_bi_dag.v
2	Unionfind using struct Node	unionfind.c	verif_unionfind.v
			verif_unionfind_rank.v
			verif_unionfind_slim.v
3	Unionfind using an array	unionfind_arr.c	verif_unionfind_arr.v
4	Unionfind using iter	unionfind_iter.c	verif_unionfind_iter.v
			verif_unionfind_iter_rank.v
5	Disposing a bigraph	dispose_bi.c	verif_dispose_bi.v
6	Garbage collector	gc.c	verif_garbage_collect.v

With the exception of the last row, all the .c and .v files are in the directory `RamifyCoq_VST/RamifyCoq/sample_mark/`.
The garbage collector algorithm was sufficiently involved that we placed its files in a separate directory, `RamifyCoq_VST/RamifyCoq/CertiGC/`.

As the table shows, we verified some C programs repeatedly, *i.e.* using different Coq specifications. For instance, we verified `mark_bi.c` by abstracting the problem to a mathematical bigraph (`verif_mark_bi.v`) and also by abstracting the problem to a mathematical directed acyclic graph (`verif_mark_bi_dag.v`).

The artifact supports the claims made in the paper in that it does actually verify six algorithms, as summarised in the table above. The mathematical graph model described in §4 of the paper is built over several files in the directory `RamifyCoq_VST/RamifyCoq/graph` and the spatial graph explored in §5 is in `RamifyCoq_VST/RamifyCoq/msl_application`.

III: Coq + Emacs + ProofGeneral Guide

For those unfamiliar with Coq, Emacs, and ProofGeneral, we provide a guided to opening, exploring, and understanding the verification of unionfind inside our Docker build. Here we explain Emacs commands as a+b, c+d. By this we mean four keystrokes: "hold a and type b, and then hold c and type d". The plus and the comma are meant for readability and are not to be typed.

1. After entering our Docker machine, type emacs to start Emacs.

2. To open a file, type Ctrl+x, Ctrl+f. This will enter you into "find file" mode, and you will see a prompt on the bottom left asking you for a file name. At the prompt, key in ~/RamifyCoq_VST/RamifyCoq/sample_mark/verif_unionfind.v.

3. In the Docker machine, we have installed ProofGeneral, which is a plugin into Emacs that arms the simple text editor with additional proof-specific features. Since you just opened a Coq file (i.e. with a .v extension), ProofGeneral will automatically kick into action in "coq mode".

4. Now you can use ProofGeneral's commands to navigate the proof. In particular:
   Ctrl+c, Ctrl+n makes the editor "step through" the next line of the proof in a REPL style.
   Ctrl+c, Ctrl+u reverses this, retracting by one line.
   Ctrl+c, Ctrl+b steps through the entire file (warning, lengthy step).
   Ctrl+c, Ctrl+RET steps until whichever line the cursor is on.

5. When a particular line of code gets underlined and there are no complaints from ProofGeneral, that means that the commands/tactics on that line of code were accepted happily by Coq.

6. We will often see Lemma <NAME>: <STATEMENT>. Proof. <TACTICS>. Qed.
   The assertion here is that the TACTICS following Proof will prove the lemma's STATEMENT. This assertion is checked by the command Qed. So if we are able to "step through" until Qed without complaint from Coq, we know that the lemma was proved.

7. The key proof in this example is Lemma body_find starting on line 183. Its statement is a little obscure, but it is saying that the function find (f_find from our C code) conforms to the specification we defined for it (find_spec from line 43 of the file verif_unionfind.v).

8. find_spec combines definitions and relations defined in other parts of our development. In general, to dig a little deeper and see any definition more fully, users can move the cursor to the definition in question and type Ctrl-c, Ctrl-a, Ctrl-p, RET. This prints out the definition. Alternately, users can type Ctrl-c, Ctrl-a, Ctrl-p and then type out the name of the definition they are interested in, followed by RET. A little investigation of find_spec shows that this corresponds to the specification we claimed in Fig 1 of the paper.

9. To exit Emacs, type Ctrl-x, Ctrl-c. You may be prompted to save changes to the file (we recommend not editing our files) and may be warned about exiting while active processes are running (this is okay, you can type "yes"). This will bring you back to the Docker machine's command line prompt. To exit the Docker machine and go back to your own machine, type exit.

This guide can be extended to our other examples by substituting the name of the verif_ file in step 2 above. Please refer to the table on page 2 to see what the relevant file names are. Please note that the files pertaining to the garbage collector are in a separate directory, i.e. RamifyCoq_VST/RamifyCoq/CertiGC/.