`dilemma` Artifact Evaluation Instructions

Introduction

This artifact includes instructions for running a docker image which contains our implementation of our lemma synthesis technique, as well as all of the tests that we ran and scripts/instructions to run them. The artifact includes a script that compiles all of the tests and generates the lists of lemmas that were synthesized.

There are 3 sets of benchmarks we considered: VFA, clam, and lia. We'll list the breakdowns of the successes and failures for those sets. Instructions/information for how we categorized a result to be a success, found a useful lemma or fail are described in greater detail within the step-by-step evaluation portion below.

- VFA Benchmarks (226): 77 Successes, 20 Usefuls, 129 Failures
- Lia Benchmarks (38): 14 Successes, 3 Useful, 21 Failures
- Clam Benchmarks (171): 60 Successes, 111 Failures

There are about 20 hours worth of tests to run, we've broken these down into approximately 20 minutes groups so that you can run some and come back to others later. The evaluation of the results requires some manual effort (looking at the top 5 lemmas returned and comparing them to the target). There are more instructions and details on these later on in the document.

Hardware Requirements

There shouldn't be any special hardware requirements. All of our experiments were run on MacOS using an Apple M1 Pro chip with 10 cores and 16GB memory. Our tool does run multiple processes concurrently, but there are no requirements. We use as many cores that are available and if there are none, then the code is run linearly. We expect any laptop or computer that any evaluator has should work.

Getting Started

Setting up Artifact

Download the file `dilemma-docker.tar` that is provided. The following commands will start-up the virtual environment, see the following sections for exact commands to run to evaluate:

```
$ docker load -i {...}/dilemma-docker.tar
$ docker run -ti -v ${PWD}/results:/root/dilemma-artifact/results
dilemma-artifact
                        <starts environment>
$ root@bf512e7c455b:~/dilemma-artifact# bash make_all.sh
$ root@bf512e7c455b:~/dilemma-artifact# eval $(opam config env)
```

At this point, the environment is set up to run. The "`make_all.sh`" script ensures that all the dependencies for the tests are in place and installed. Finally, "`eval $(opam config env)`" ensures that the command line is able to find all path variables (specifically, for our evaluation that means the command "coqc" is available). The following sections will detail experiment setup and evaluation instructions.

Note, all of the results from the evaluations are expected to be found in the `${PWD}/results` directory as specified in the run command. That is, after running the container there should be a directory created in that same directory titled `'results'`.

Note: Ensure that you run the `docker run` command outside of your downloads folder. In general, there are some folders that docker is not able to modify (in our case, add a results folder). We've had success running the command from the desktop folder or home folder on your local machine.

Simple Test (Kick-the-Tires)

There are more details in the experiment set up – specifically, with respect to what the benchmarks are and how to evaluate the results. In this section, we'll make sure you can run a single group and a single test suite. The remaining portion of the artifact evaluation will entail running the remaining groups/suites and then reviewing the results.

There are 14 sets of benchmarks that we are currently evaluating on. For each of the benchmark suites, we've broken them into groups where each group should take approximately 20 minutes to run.

Run a *Benchmark Suite*: Try running the bagperm benchmark suite. Following the set up listed above, run the following command:

```
$ root@bf512e7c455b:~/dilemma-artifact# bash run.sh bagperm
```

***Benchmark Suite* Result:** Following the command above, there should be a file titled "suite_bagperm.txt" found in the results folder that is generated from the `docker run` command. This file should include various outputs for each test (we'll explain how to interpret these results later in this document) that looks something like this:

```
-------------------------------------------------------
```

```
Test: bag_perm_by_bag_eqv_uncons
Target: bag_eqv (b :: l1) (b :: l2) -> bag_eqv l1 l2

(bag_eqv (n :: al) (n :: bl) -> bag_eqv al bl)

Number of Result Pairs Returned (before QuickChick): 1

Time Elapsed From Start: 29.164 seconds

--------------------------------------------------------

--------------------------------------------------------
Test: bag_perm_by_count_insert_other
Target: y <> x -> count y (l1 ++ x :: l2) = count y (l1 ++ l2)

(count n0 (a :: al) = count n0 (n :: x ++ a :: x0) ->  count n0
al = count n0 ((n :: x) ++ x0))

Number of Result Pairs Returned (before QuickChick): 1

Time Elapsed From Start: 102.255 seconds

--------------------------------------------------------

--------------------------------------------------------
...file continues...
```

Run a *Benchmark Group*: Try running the first selection benchmark group (selection_2). Following the set up listed above, run the following command:

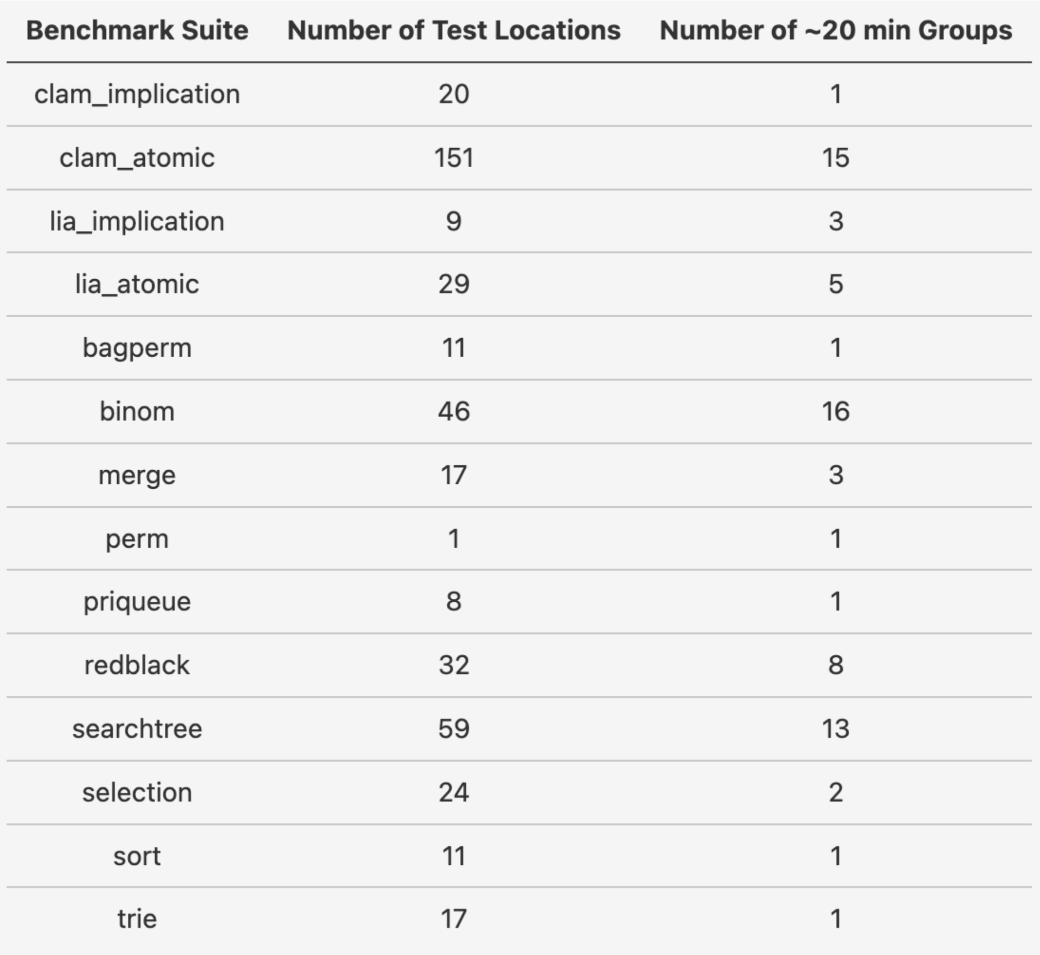
```
$ root@bf512e7c455b:~/dilemma-artifact# bash run.sh group selection_2
```

***Benchmark Group* Result:** Following the command above, there should be a file titled "group_selection_2.txt" found in the results folder that is generated from the `docker run` command. This file should include various outputs similar to one above.

If you are able to run both of these commands, you should be good to run all of them. We would suggest if you take breaks between running to make sure you have the results saved in a different directory (in order to prevent any overwriting of files and needing to rerun test sets).

Experiment Setup

There are 14 sets of benchmarks that we are currently evaluating on. Based on reviewers' feedback and visions, we expect that approximately 2 more sets of benchmarks will be added. The table below details the benchmarks currently accounted for in this artifact:



Benchmark Suite	Number of Test Locations	Number of ~20 min Groups
clam_implication	20	1
clam_atomic	151	15
lia_implication	9	3
lia_atomic	29	5
bagperm	11	1
binom	46	16
merge	17	3
perm	1	1
priqueue	8	1
redblack	32	8
searchtree	59	13
selection	24	2
sort	11	1
trie	17	1

We've provided two means of running the tests. The first option is to run a benchmark suite at a time. In many cases this is probably easiest, however, there are a few larger benchmarks that we have divided into groups to make it easier to run groups at a time and come back later. The rightmost column details how many groups that benchmark was divided into.

Note, that we cut off each group at ~20 minutes, however this was done greedily. So, there are cases where the full 20 minutes is not taken, and in cases where a single test might take longer than 20 minutes that group will just hold that single test. The different groups from each benchmark and their expected runtimes are listed below:

Benchmark Suite	Group Labels	Expected Runtime (minutes)
clam_implication	clam_implication	22
clam_atomic	clam_atomic_1	21
	clam_atomic_2	20
	clam_atomic_3	15
	clam_atomic_4	21
	clam_atomic_5	18
	clam_atomic_6	21
	clam_atomic_7	13
	clam_atomic_8	21
	clam_atomic_9	16
	clam_atomic_10	29
	clam_atomic_11	21
	clam_atomic_12	21
	clam_atomic_13	20
	clam_atomic_14	3
	clam_atomic_15	-
lia_implication	lia_implication_1	16
	lia_implication_2	46
	lia_implication_3	20
lia_atomic	lia_atomic_1	20
	lia_atomic_2	20
	lia_atomic_3	19
	lia_atomic_4	20
	lia_atomic_5	7
bagperm	bagperm	12
binom	binom_1	20
	binom_2	20
	binom_3	20
	binom_4	16
	binom_5	21
	binom_6	19
	binom_7	19
	binom_8	38
	binom_9	18
	binom_10	20
	binom_11	6
	binom_12	17
	binom_13	38
	binom_14	15
	binom_15	29

	binom_16	4
merge	merge_1 merge_2 merge_3	16 20 7
perm	perm	1
priqueue	priqueue	18
redblack	redblack_1 redblack_2 redblack_3 redblack_4 redblack_5 redblack_6 redblack_7 redblack_8	11 22 16 20 21 15 18 19
searchtree	searchtree_1 searchtree_2 searchtree_3 searchtree_4 searchtree_5 searchtree_6 searchtree_7 searchtree_8 searchtree_9 searchtree_10 searchtree_11 searchtree_12 searchtree_13	19 19 19 19 6 17 17 13 21 13 19 17 10
selection	selection_1 selection_2	20 9
sort	sort	12
trie	trie	18

** *NOTE:* ***clam_atomic_15*** *contains clam benchmarks that we expect to fail to terminate either because they have too large of a search space time out and/or throw an error. We've marked these tests as failures in our evaluation. Because they terminate with failure, we didn't keep track of timing.*

Each benchmark suite or group contains a set of files, each file is a Coq file which includes a partial proof where the proof is ended with a call to our tactic (`dilemma. Admitted.`). To run each test individually, one can call into the directory and compile the file. For example to run the test `select_rest_length_by_select_perm.v` from the selection benchmark suite, one would run:

```
$ cd dilemma-artifact/benchmarks/vfa_selection/tests
$ coqc select_rest_length_by_select_perm.v
```

The results will be available printed out to the terminal (and the whole log can be seen in the same folder in a folder titled "`log_for_select_rest_length_by_select_perm1.txt`".

We've set up two means of running sets of benchmarks both using the `run.sh` script: (1) run the whole benchmark suite at a time and (2) run a group at a time (restricted expected runtime). The next section will detail evaluation instructions.

** NOTE: Sometimes when running tests back to back, the temporary files we write get messed up due to space issues (especially in the docker image). Oftentimes if you rerun the experiment it will work. In these cases if there are no results in the log after running (that is just the test and target lemma are listed with no info), you can either rerun the tests that didn't complete in each group individually or let us know which tests didn't complete and we can make another group of those tests to rerun (and push those changes to the repo). If rerunning again fails, then the case should be marked as failure.

To test individually, just move to the directory where that test is located and compile that file. For example, suppose the test `t.v` from the benchmark set `benchmark_suite` needs to be reran you can run the following commands:

```
$ root@bf512e7c455b:~/dilemma-artifact# cd benchmarks/benchmark_suite/tests
$ … :~/dilemma-artifact/benchmarks/benchmark_suite/tests# coqc t.v
```

The response that matches the output found in the results file should be printed to the terminal (that is the top 5 suggested lemmas).

This should not occur too frequently - although it is likely that running the docker image will cause these issues to be more common. Let us know if the issues persist and we can break the groups down more so the tests are less cumbersome and/or make groups of tests that need to be reran so we can put them all under one group label to make it easier. For reference, there were only a few cases that needed to be rerun in the VFA benchmarks during our evaluation, and a bit more in the lia/clam benchmarks due to search complexity of those benchmarks.

Step-by-Step Instructions (Evaluation Instruction)

Generate Results

To run the whole benchmark suite at time run:

```
$ root@bf512e7c455b:~/dilemma-artifact# bash run.sh <benchmark>
```

Where `<benchmark>` is the name of the benchmark suite to run (should correspond to a row in the first table listed in this document). For example, to run the redblack benchmark suite, you would run:

```
$ root@bf512e7c455b:~/dilemma-artifact# bash run.sh redblack
```

The `results` folder will hold the results from this benchmark suite in a file called "`suite_redblack.txt`". The same is expected for running any of the suites (where file is labeled suite_<suite name>.txt).

To run the a single group of benchmarks at time run:

```
$ root@bf512e7c455b:~/dilemma-artifact# bash run.sh group <group>
```

Where `<group>` is the name of the group of benchmarks to run (this should be an element from the middle column of the second table listed above). For example, to run the second group from the `merge` benchmark suite, you would run:

```
$ root@bf512e7c455b:~/dilemma-artifact# bash run.sh group merge_2
```

The `results` folder will hold the results from this benchmark suite in a file called "`group_merge_2.txt`". The same is expected for running any of the groups (where file is labeled group_<group name>.txt).

All results will be generated once all of the benchmark suites have been run. This will either happen by running the benchmark suite as instructed above, or by running all of the groups found within that benchmark suite.

The results generated are the lemmas that we've synthesized – the following section details how these results should be interpreted/evaluated.

Interpret Results

The snip below is from one of the result files:

...

```
--------------------------------------------------
```

```
-------------------------------------------------------
Test: select_rest_length_by_select_perm
Target: select x l = (y, r) -> Permutation (x :: l) (y :: r)

(select x l = (y, r) -> length l = length r)

(select x l = (y, r) -> Permutation (y :: r) (x :: l))
(Permutation (y :: r) (x :: l) -> length l = length r)

(select x l = (y, r) -> Permutation (x :: l) (y :: r))
(Permutation (x :: l) (y :: r) -> length l = length r)

Number of Result Pairs Returned (before QuickChick): 3

Time Elapsed From Start: 58.977 seconds
-------------------------------------------------------
-------------------------------------------------------
```
…

Each result file will have one of these sections for each test that is run. The first line indicates the file that the test is stored in, and the second line represents the helper lemma that was used at the location. The following lines are the lemmas that we've synthesized (this will be restricted to the top 5 for each test). The following two lines are not relevant for evaluation (one notes the amount of results generated and the other is the runtime).

In the case no lemmas or statistics are returned, that means that the test didn't finish running. Sometimes there are resource issues that occur from running a bunch of tests in a row, you can rerun these tests (described above). Again, note that the `clam_atomic_15` group specifically includes tests that we expect to not terminate.

In order to interpret the results, you'll need to manually look at each result. The potential categories a test might fall under are:

- Success – we synthesized the target lemma
    - An exact match to target (variables most likely differ)
    - An exact match to target modulo reflexivity
    - Implication has a precondition which is an equality and if that was written in the goal of the implication and that matches the target lemma, then this is also a success. Suppose the target lemma was `Permutation (x :: l) (y :: r)` and we generated `a = x :: l → Permutation a (y :: r)`, this would be counted.

- Useful – we synthesized a useful lemma
    - A lemma found is used later in the same proof, so we found a lemma that was used at some point to complete the proof. This is considered useful. This requires looking at the file to see what lemmas were used later in the proof.
    - Successfully found one precondition, but still need to weaken other assumptions to match exactly. For example, suppose the target lemma is A → B → C and we synthesize A → D → C. This is deemed useful because we've successfully isolated one needed precondition (weakened one assumption).
    - We've found a lemma that is weaker than the target lemma but is able to be used in the same way. For example, if the target lemma is `In x l` ⋁ `In x m → In x (l ++ m)` and we find `In x l → In x (l ++ m)`, where is lemma can be used instead of the stronger target lemma. This is considered useful.
- Fail – test is out of scope, no results generated, or no useful results are generated

We've included the logs from the evaluations that we ran for our paper in the artifact as well. Within the directory `paper_results/processed_results`, there are files for each of the benchmarks containing the categorization we've assigned. These contain the same information from the results you should have generated (some of the notation is slightly different). These files also include all of the categorization notes that we've included from our analysis.

All of the other folders within `paper_results` hold the raw files that are generated as a byproduct of running our tool. These files should be generated in the docker container as you're running the tests - these will not be moved to the results folder, so if you want to look at them you'll do so through the docker container environment and look in the folders where the tests are stored. To complete the evaluation, there shouldn't be a need to look at these files but you can if you are curious.

Expected Results

As cited in our paper, we expect the following breakdown of results…

VFA Benchmarks (226)

- 77 Successes
- 20 Usefuls
- 129 Failures

Lia Benchmarks (38)

- 14 Successes
- 3 Useful
- 21 Failures

Clam Benchmarks (171)

- 60 Successes
- 111 Failures

Note, there is a bit of non-determinism in our procedure for example generation, so it is possible that the results won't match exactly. For example, if insufficient examples were generated or some resource issue occurs, reducing the proof state and/or synthesizing the preconditions might not behave as expected.

Advice for Completing

The total runtime expected to run all of the tests is about 20 hours. We've broken it down so you can do a few chunks at a time and just let it run in the background of whatever you're doing. You can also obviously analyze the results individually before they are all done.

Note, if you run a suite or a group and stop running before it has finished, you won't be able to see the results since they are printed out in batches. This is why we had opted to break the groups down so that you can do it incrementally. If it would help to have them broken down more, let us know and we can add more groups for you to run.

Changes Expected From Revisions

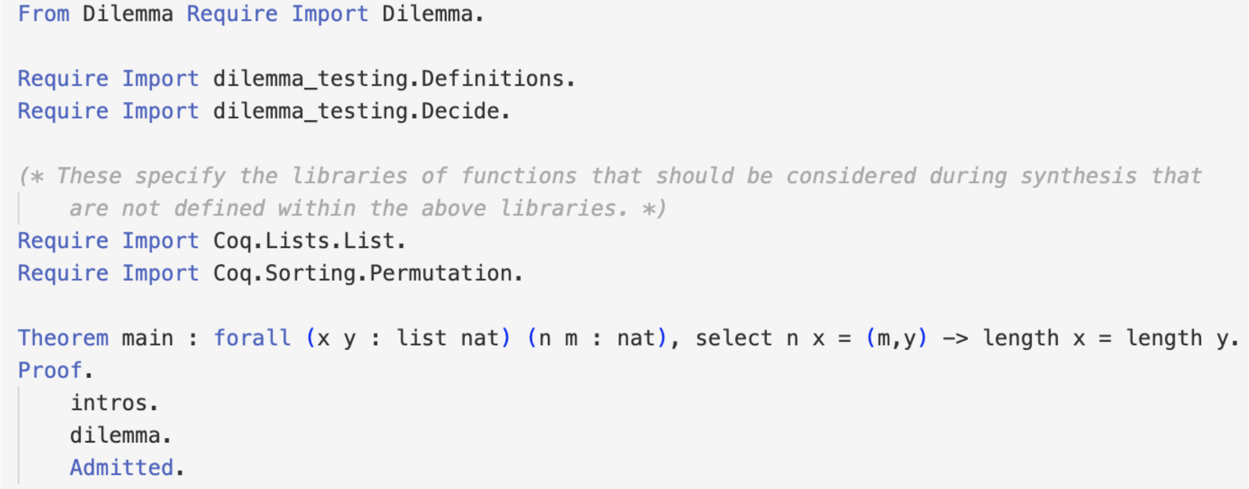
There are two changes that might occur/are expected to occur in the artifact resulting from the revisions suggested by reviewers.

1. More tests will be added. Specifically, we expect that two benchmark suites will be added (each potentially divided into an implication and atomic lemma group).
2. Change to implementation to improve the runtime. Any changes made at this point would be made to improve the runtime and will not be made if any substantial changes to results are caused by the change.

Reusability Guide

Our artifact is a Coq tactic. The only requirements beyond the installation requirements that are needed to use our tool are proofs of decidability for any data types and propositions that are in the scope of the proof.

In the docker image, the folder \`examples\` contains two examples of our tactic being invoked. The files \``selection_e1.v`\` and \``selection_e2.v`\` both showcase the tactic being used. This is the \``selection_e1.v`\` file:



```
From Dilemma Require Import Dilemma.

Require Import dilemma_testing.Definitions.
Require Import dilemma_testing.Decide.

(* These specify the libraries of functions that should be considered during synthesis that
    are not defined within the above libraries. *)
Require Import Coq.Lists.List.
Require Import Coq.Sorting.Permutation.

Theorem main : forall (x y : list nat) (n m : nat), select n x = (m,y) -> length x = length y.
Proof.
    intros.
    dilemma.
    Admitted.
```

The file “dilemma_testing.Definitions” includes the relevant definitions for the proof. The file “dilemma_testing.Decide” contains the proofs of decidability for the propositions and equality decidability for types. We list `Require Import` with the modules whose definitions we want to be considered in our synthesis (that are not defined in the other imported modules). Note, we only consider definitions that are defined in the imported modules not imported into those modules.

In order to invoke our tool, you just call the `dilemma` tactic, followed by admitting the proof and then compile the file with `coqc`.

Besides runtime, the main limitation of our tactic is the requirement of needing proofs of decidability. While in practice, these are somewhat routine and can often be automatically found; this is not always the case.