Artifact Guide: Data-Driven Abductive Inference of Library Specifications

This document describes installation and use of Elrond, an OCaml tool for data-driven abduction of black-box library method specifications. This is the accompanying artifact for the OOPSLA 2021 submission *Data-Driven Abductive Inference of Library Specifications* by Zhou et al.

Claims Supported / Not Supported By This Artifact

The paper's evaluation section discusses 5 key research questions the evaluation is designed to address.

- *Q1: Is Elrond able to find specifications sufficient to verify a range of properties and client programs in a reasonable amount of time?*
  - This artifact supports the claims that our tool covers a rich set of data types, predicates, and properties drawn from the literature over a diverse set of client programs. The library and client use cases discussed in the paper are found in the `data` directory of this artifact.
  - This artifact may not support the claim that Elrond can infer a consistent initial solution in under 3 minutes (line 1004) depending on the hardware on which the benchmarks are executed, although in our testing this claim was verified.
- *Q2: Can Elrond identify unsafe client programs?*
  - This artifact supports the claim that Elrond can return concrete counter-examples for clients with unsafe postconditions by providing 5 benchmarks containing unsafe client programs.
- *Q3: Can Elrond efficiently find maximal solutions?*
  - This artifact may not support the claim that `16/22` benchmarks of our safe benchmarks were able to find maximal solutions from the initial solution within `1` hour (lines `1015-1016`) depending on the hardware used to run the benchmarks. However, this claim held in our testing.
- *Q4: Is Elrond able to find useful intermediate generalizations of initial specifications?*
  - This artifact supports the claim that Elrond considers at most 40% of the full search space (line `1023`) by providing results similar to the `#Gather/|phi+|` column in Table 4.
- *Q5: Does weakening improve the quality of the inferred specifications?*
  - The artifact supports the claim that `time_d` of short benchmarks are samller than long benchmarks (line `1037-1040`) by providing results similar to the `time_d` column in Table 4.

As we claimed in the paper (lines `1066-1067`), we can verify most of our results except 4 specifications which are commented out in the Coq makefile located in `proof/_CoqProject`.

Requirements

- Docker, which may be installed according to the official installation instructions. This guide was tested using Docker version 20.10.7, but any contemporary Docker version is expected to work.
- We recommend machines have at least 16 GB of memory and 8 GB of hard disk space available when building and running Docker images. These benchmarks were tested on two desktop machines, an Intel i7-6700K CPU @ 4.00GHz with 16 GB RAM and an Intel i7-8700 CPU @ 3.20GHz with 64GB of RAM.

Getting Started

This artifact is built as a Docker image. Before proceeding, ensure Docker is installed. (On *nix, `sudo docker run hello-world` will test your installation.) If Docker is not installed, install it via the official installation guide.

Using the Pre-Built Docker Image

You may fetch the pre-built Docker image from Docker Hub:

```
# docker pull elrondinfer/elrond:2021-oopsla
```

Building the Docker Image

Alternately, to build the Docker image yourself, navigate to the directory containing the Dockerfile and tell Docker to build:

```
# docker build . --tag elrondinfer/elrond:2021-oopsla
```

Running the Docker Image

To launch a shell in the Elrond Docker image, say:

```
# docker run -it elrondinfer/elrond:2021-oopsla
```

You can print Elrond's help message to verify the tool is operating successfully:

```
$ ./main.exe --help
```

The `bankersq` benchmark should succeed in under 10 seconds (depending on hardware):

```
$ ./main.exe infer full data/bankersq.ml data/bankersq_assertion1.ml bankersq_out
```

When you are finished using the image, you may stop it by terminating the shell with `exit`.

Running Benchmarks

Comprehensive Scripts

The following scripts run the benchmark suite displayed in Table 4 of the paper:

- `./bin/run_benchmarks_short.sh` executes all shorter-running benchmarks (~1.5 hours and ~5 hours on our two test machines).
- `./bin/run_benchmarks_long.sh` executes all longer-running benchmarks (> 10 hours on our test machines).
- `./bin/visualize_running_result.sh` visualizes from results which were just run (immediate).

Detailed Steps

The above scripts automate the following process:

1. Use Elrond to find consistent (but not weakened) specification mappings:

   ```
   $ python3 bin/evaluation_tool.py consistent config/standard.config
   ```

   Output of these mappings is stored in the `_benchmark_<name>` directories.

2. Use Elrond to weaken the consistent specifications. There are six benchmarks labeled as `Limit` in Table 4 which take more than one hour to complete. We have grouped these long-running benchmarks separately; you may execute the shorter-running weakening benchmarks with:

   ```
   $ python3 bin/evaluation_tool.py weakening config/standard.config -s
   ```

   and the longer-running weakening benchmarks with:

   ```
   $ python3 bin/evaluation_tool.py weakening config/standard.config -l
   ```

   Output is again stored in the `_benchmark_<name>` directories.

3. Calculate the times needed for the SMT solver to find a sample allowed by a weakened solution but not the initial one (`time_d` in Table 4.) This calculation is not a core part of Elrond, but rather a post-processing step for our experimental evaluation.

   ```
   $ python3 bin/evaluation_tool.py diff config/standard.config
   ```

4. Count the total positive feature vectors in the space of weakenings (`|phi+|` in Table 4). Again, this is a post-processing step for our experimental evaluation rather than a core part of Elrond.

   There are three cells in Table 4 shown in blue, indicating the benchmarks timed out in our testing. You can perform the counting for all benchmarks except these three which timed out with:

   ```
   $ python3 bin/evaluation_tool.py count config/standard.config -s
   ```

   To run the counting for the three benchmarks which timed out for us, say:

   ```
   $ python3 bin/evaluation_tool.py count config/standard.config -l
   ```

   The latter command may obviously take a long time to complete.

5. Build the table:

   ```
   $ python3 bin/evaluation_tool.py table config/standard.config
   ```

The table may be displayed at any stage of the benchmark process; any missing entries will be displayed as `None`.

Building Figure 5

`python3 bin/evaluation_tool.py figure config/prebuilt.config` generates Figure 5 from the prebuilt weakening experimental results. (After running the benchmarks locally, this figure may be generated with the `config/standard.config` configuration instead.) The figure is saved in the `_result` directory.

Building From Saved Results

`./bin/visualize_prebuilt_result.sh` visualizes from prebuilt results (immediate).

Running Elrond

Elrond requires both a source file and assertion file as input, and outputs results in JSON format to some output directory. The input source and assertion files for the benchmark suite are located in the `~/ADT-Lemma-Discovery/result` directory in the Docker image.

The command to run an individual input without weakening is:

```
$ ./main.exe infer consistent <source_file> <assertion_file> <output_dir>
```

For example,

```
$ ./main.exe infer consistent data/bankersq.ml data/bankersq_assertion1.ml bankersq_out
```

will run the `bankersq` benchmark, writing results to the `_bankersq_out` directory.

When the assertion is unsound, Elrond will print the counter-example it found. The following command executes a benchmark with an incorrect assertion:

```
$ ./main.exe infer consistent data/customstk.ml data/customstk_assertion2.ml customstk_out
```

Elrond returns the following counterexample:

```
CEX:
s2 -> [0],s1 -> [],il_0 -> [0]
Failed with Cex in 0.088947(s)!
```

To find weakened specification mappings, first run the benchmark without weakening as above, then say:

```
$ ./main.exe infer weakening <output_dir>
```

on the same `<output_dir>`.

Note that the weakening may take long time to run, `-tb <time bound>` sets the time bound(in seconds) for weakening inference. The default time bound used by benchmark scripts is 3600 seconds.

For example,

```
$ ./main.exe infer weakening bankersq_out -tb 3600
```

will perform weakening on the `bankersq` benchmark we executed above with a 3600 second time bound.

You may run the full inference-with-weakening pipeline at once by saying:

```
$ ./main.exe infer full <source_file> <assertion_file> <output_dir>
```

For example, we can recreate the `bankersq` output directory in one pass:

```
$ rm -rf _bankersq_out
$ ./main.exe infer full data/bankersq.ml data/bankersq_assertion1.ml bankersq_out
```

Displaying Specification Mappings

The following command displays inferred specifications before weakening:

```
$ ./main.exe show consistent <output_dir>
```

where `<output_dir>` is the location of Elrond inference output.

For example, to infer and display specifications from the paper's motivating example (called `exampleout` in this artifact), run the following commands:

```
$ ./main.exe infer full data/customstk.ml data/customstk_assertion1.ml exampleout -sb 4
$ ./main.exe show consistent exampleout
```

(Here, the `-sb 4` flag limits the sampling bound to small number in order to simulate a biased test generator.)

This yields the following result:

```
Customstk.push(i_0,il_0,il_1):=
forall u_0,(ite mem(il_1,u_0)
    (ite mem(il_0,u_0)
        (
         !(u_0==i_0) &&
         !hd(il_1,u_0)
        )
        ((u_0==i_0) && hd(il_1,i_0)))
    (
     !(u_0==i_0) &&
     (
      !mem(il_0,u_0) &&
      (
       !hd(il_0,u_0) &&
       !hd(il_1,u_0)
      )
     )
    ))
```

The following command displays the weakened specification mappings (`-s` asks for simplified specifications):

```
$ ./main.exe show weakening <output_dir> -s
```

For example, the following command displays weakened specifications for the paper's motivating example (assuming the specifications have already been found as above):

```
$ ./main.exe show weakening exampleout -s
```

This yields the output:

```
Customstk.push(i_0,il_0,il_1):=
forall u_0,(ite mem(il_1,u_0)
```

```
    (
     (u_0==i_0) ||
     (
      mem(il_0,u_0) &&
      !hd(il_1,u_0)
     )
    )
    (
     !(u_0==i_0) &&
     (
      !mem(il_0,u_0) &&
      (
       hd(il_0,u_0) ||
       !hd(il_1,u_0)
      )
     )
    ))
```

Input File Formats

Elrond expects both an input OCaml code listing and an assertion file. The input code listing is given as a specially formatted OCaml source file with certain restrictions, and looks as follows:

```
(* The library signature. *)
module type DT_NAME = sig
  type TP_NAME
  ...
  val VAR: FUNC_TP
  ...
end

(* The type of the client function. *)
val VAR: FUNC_TP

(* The client implementation. *)
let [rec] VAR (VAR: ARG_TP) ... = EXPR
```

The all-caps placeholders are filled according to the following grammar:

```
DT_NAME:= string
TP_NAME:= string | DT_NAME "." TP_NAME
ARG_TP:= "int" | "bool" | TP_NAME
RET_TP:= "int" | "bool" | TP_NAME | "(" FUNC_TP "," ... ")"
FUNC_TP:= RET_TP | ARG_TP "->" FUNC_TP

VAR := string
VAR_TUPLE := VAR | "(" VAR "," ... ")"
Lit := integer | boolean
OP := "+" | "<=" | ">=" | ">" | "<" | "=="
FUNC_APP := FUNC_NAME | FUNC_APP VAR
CASE := "| _ when" EXPR "->" EXPR
EXPR :=
| "if" FUNC_APP "then" EXPR "else" EXPR
| VAR
| EXPR OP EXPR
| "(" EXPR "," ... ")"
| FUNC_NAME
```

```
| EXPR EXPR
| "let" VAR_TUPLE : ARG_TP "=" EXPR "in" EXPR
| match VAR_TUPLE with CASE ...
```

The input source file must observe the following rules:

- The type in signature must be abstract.
- The input type of a function is a list of `ARG_TP`, and the output type of a function is written as a tuple.
- The condition of an `if` statement must be a single function application.
- The match cases in `match` statements must be written as `| _ -> when CASE` instead of `| CASE`. (This is because we use the OCaml parser, which asks that the matched case be an application of a datatype constructor. However, here the type is abstract.)
- New variables must be typed when they first appear (we do not perform any type inference).
- All variables must have distinct names (we do not do perform any alpha renaming).

The assertion file is formatted as follows:

```
(* Predicates *)
let preds = [| PRED; ...|]

(* Precondtion (optional) *)
let pre VAR (IVAR: ARG_TP) ... (OVAR: ARG_TP) ... (QVAE: ARG_TP) ... = ASSERTION
(* Postcondtion *)
let post VAR (IVAR: ARG_TP) ... (OVAR: ARG_TP) ... (QVAE: ARG_TP) ... = ASSERTION
```

where the all-caps placeholders are filled according to the following grammar:

```
DT_NAME:= string
TP_NAME:= string | DT_NAME "." TP_NAME
ARG_TP:= "int" | "bool" | TP_NAME

IVAR := string
OVAR := string
QVAR := string

PRED := "mem" | "hd" | "ord" | "once" | "left" | "right" | "para" | "ance" | "root"
OP := "==" | "!=" | "<=" | ">=" | "<" | ">"

ASSERTION :=
| PRED VAR ...
| VAR OP VAR
| implies ASSERTION ASSERTION
| iff ASSERTION ASSERTION
| ASSERTION "&&" ASSERTION
| ASSERTION "||" ASSERTION
| NOT ASSERTION
```

Currently, implementations of libraries and predicates are fixed; users can define their own assertions, but not their own libraries or predicates.

Coq Formalization

The Coq proofs of our inferred specifications are located in the `proof` directory. These proofs may be executed by running `make`. Each file with prefix `Verify` contains the proof of one inferred specification.

Proof obligations expressed in Coq may be generated via:

```
$ dune exec -- main/main.exe coq  <specificaion mapping file> <function name>
```

This command converts the inferred specification mappings to the Coq lemmas for manual proof. For example, running:

```
$ dune exec -- main/main.exe coq _result/_customstk1/_oracle_maximal.json Customstk.push
```

prints several lemmas:

```
Lemma Customstk.push_1 (i_0:nat) (il_0:list nat) (il_1:list nat) (u_0:nat):
  (push_spec i_0 il_0 il_1) ->
  (((not (u_0 = i_0))/\(list_member  il_1 u_0)) -> (not (list_head  il_1 u_0))).
```

Here, `push_spec` is the pre-defined specification of `push` (see `proof/.*Aux.v*`) and `(((not (u_0 = i_0))/\(list_member il_1 u_0)) -> (not (list_head il_1 u_0)))` is one branch of the inferred decision tree. Proving all given lemmas establishes the correctness of the inferred specifications.

Artifact Structure

This section gives a brief overview of the files in this artifact.

- `bin/`: various scripts for collecting and displaying experimental results.
- `config/`: the configuration files for the benchmark scripts.
- `data/`: the benchmark input files.
  - `data/result.zip`: a collection of saved inference results.
- `frontend/`: the Elrond parser, a modified OCaml parser.
- `inference/`: the specification mapping inference modules, including both consistent inference and weakening inference.
- `language/`: Elrond's intermediate specification language.
- `ml/`: the decistion tree learner.
- `main/main.ml`: the main entry point of Elrond.
- `pred/`: built-in implementation of predicates.
- `proof/`: the coq proof generator.
- `solver/`: the Z3 (SMT solver) wrapper.
- `tp/`: Elrond's built-in types.
- `translate/`: VC generation.
  - `translate/imps.ml`: built-in implementation of libraries.
- `utils/`: miscellaneous utility functions.