OOPSLA 2022 Artifact (README.md)

Overview

This Readme file contains two sections. The **Getting Started** section gives the main steps for installing the dependencies using OCaml package manager (opam) followed by introducing small scripts for building and running Cobal on a sample test case.

The second section **Step-by-step Instructions** explains the structure of the directory, how to run Cobalt on all the benchmarks in the paper?, understanding the results and how to run Cobal on an individual synthesis task in different ablation modes?

Getting Started

This document is an overview of the Artifact for the *Cobalt* tool associated with the submission *Specification-Guided Component-Based Synthesis from Effectful Libraries.* The Artifact is distributed as a source control repository link.

Following are the main steps to build and run Cobalt:

Building Cobalt from Sources

We have successfully tried building Cobalt on Linux (Ubuntu 16.04) and Mac(macOs Monterey). We discuss the instructions for the Ubuntu build and running.

Prerequisites

To build Cobalt following dependencies must be installed:

- OCaml (Version >= 4.03)

```
#install opam
$ apt-get install opam

#environment setup
$ opam init
$ eval `opam env`

# install a specific version of the OCaml base compiler
$ opam switch create 4.03
$ eval `opam env`

# check OCaml installation
$ which ocaml
/Users/.../.opam/4.03.0/bin/ocaml

$ ocaml -version
The OCaml toplevel, version 4.03.0
```

- Z3 SMT Solver

```
$ opam install "z3>=4.7.1"
$ eval $(opam env)
```

- Menhir for parsing the specification language

```
$ opam install menhir
$ eval $(opam env)
```

- OCamlbuild version >= 0.12

```
$ opam install "ocamlbuild>=0.12"
$ eval $(opam env)
```

To Run the Evaluations.

- Python3

```
$ apt-get install python3
```

Building Cobalt

After all the dependencies are installed, Cobalt can be directly built using *ocamlbuild* using the script `build.sh` in the project root directory.

```
$ ./build.sh
```

The above build script will create a native executable `effsynth.native` in the project's root directory

Quick test: Test Running Cobalt

```
$ python3 quick_test.py
```

This should produce the following output giving running times for the four ablation cases *viz.* cobalt; fw-alone; bw-alone; no-cdcl; corresponding to 4 bars in Fig 9. (numbers may vary a little):

```
Running Varinat cobalt ./synth_tests/unit/checked/other_units/u_test3.spec
0.22user 0.09system 0:00.31elapsed 99%CPU (0avgtext+0avgdata
24536maxresident)k
0inputs+1528outputs (0major+2249minor)pagefaults 0swaps
{'cobalt': 0.22838}
...
{'cobalt': 0.22838, 'fw-alone': 0.92747, 'bw-alone': 0.226788, 'no-cdcl':
1.498713}
./synth_tests/unit/checked/other_units/u_test3, 0.23, 0.93, 0.23, 1.50, 0
```

Step-by-step Instructions

The following instructions explain:

1. The structure of the repository highlighting relevant files.
2. How to Run Cobalt to generate synthesis time Figure 9 a, b and c.
3. How to Run Cobalt on individual synthesis problem.

structure of the Artifact.

The source code for this Artifact is available at effsynth

The files and directories used in this Artifact are:

- `quick_test.py` : a script to test the successfull installation of Cobalt
- `run_benchmarks.py` : a script to run Cobalt for all the benchmarks in the paper producing results.
- `synth_tests/unit/checked/databases/**.spec` contains benchmarks in Figure 9a.
- `synth_tests/unit/checked/parsers/**.spec` conatains benhcmarks in Figure 9b.
- `synth_tests/unit/checked/imperative_ds_simple/**.spec` contains Imperative benchmakrs I1-I11 in Figure 9c.
- `synth_tests/unit/checked/vocal/**.spec` contains remianing benchmarks in Figure 9c.

Running the evaluation using a push button.

```
$ cd project_root
$ python3 run_benchmarks.py
```

Once the script terminates, it will produce a file `timings` which contains entries for each benchmark in Fig 9a, b, and c in the following format. (Showing the result for Png-simple benchmark P1). The time 1000 shows the case where the tool timed out.

```
******************************
./synth_tests/unit/checked/parsers/png_simple.spec_cobalt :
0.02336999999999989
./synth_tests/unit/checked/parsers/png_simple.spec_fw-alone :
```

```
1.5463100000000054
./synth_tests/unit/checked/parsers/png_simple.spec_bw-alone :
0.017842999999970743
./synth_tests/unit/checked/parsers/png_simple.spec_no-cdcl : 1000
 ******************************
```

Note (Exact reproducibility): Note that we have refined the implementation since the time of paper submission; thus, not all the numbers will match with those in the paper. However, the general relation between the timings of the approaches still holds for benchmarks, as visible in the above example.

Running an individual synthesis query

- To run Cobalt to get the synthesis results using the complete cobalt mode ($\textcolor{blue}{blue}$ bar in Fig 9):

```
$ effsynth.native -bi -cdcl file.spec
```

- To run Cobalt in FW-Alone mode ($\textcolor{red}{red}$ bar in Fig 9):

```
$ effsynth.native -cdcl file.spec
```

- To run Cobalt in BW-Alone mode ($\textcolor{black}{black}$ bar in Fig 9):

```
$ effsynth.native -bi file.spec
```

- To run Cobalt in FW-No-cdcl mode ($\textcolor{green}{green}$ bar in Fig 9):

```
$  effsynth.native file.spec
```