Artifact Guide: Covering All the Bases: Type-based Verification of Test Input Generators

This is the accompanying artifact for the PLDI 2023 submission *Covering All the Bases: Type-based Verification of Test Input Generators* by Zhou et al. This artifact consists of both the OCaml impelementation (**Poirot**) and the Coq formalization of the type system or our core language λ^{TG} introduced in the paper.

Getting Started Guide

We recommend machines have at least 8 GB of memory and 8 GB of hard disk space available when building and running Docker images. All benchmarks were tested on MacBook Pro 14-inc, 2021, that has an Apple M1 Pro CPU with 16 GB RAM. The estimated execition time in the rest of document also fits this setting.

Requirements

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. This guide was tested using Docker version `20.10.23`, but any contemporary Docker version is expected to work.

Using the Pre-Built Docker Image

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

```
$ docker pull poirot23/poirot:pldi-2023
```

Building the Docker Image (Optional)

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

```
$ docker build . --tag poirot23/poirot:pldi-2023
```

Resource Requirements: Although our tool **Poirot** and the Coq formalization doesn't have large memory usage, building the docker image needs more than `32GB` RAM available. This memory usage requirement comes from the then installation of the SMT solver `z3` (https://github.com/Z3Prover/z3). When the RAM limit of the Docker (by default, it is `8GB` on Mac, no limit on Linux machine) is lower than `32GB`, the installation of `z3` will be killed and the `docker build` will fail. The memory error can be fixed by increasing the RAM limit in Docker; you can find instructions for doing so on Mac here: (https://docs.docker.com/desktop/settings/mac/#resources), for Windows here: (https://docs.docker.com/desktop/settings/windows/#resources), and for Linux here: (https://docs.docker.com/desktop/settings/linux/#resources). The pre-built docker image is built on a Linux machine having Intel i7-8700 CPU @ 3.20GHz with `64GB` of RAM, it took `30` min to build.

Running the Docker Image

To launch a shell in the Docker image, say:

```
$ docker run -it -m="8g" poirot23/poirot:pldi-2023
```

To compile **Poirot**, say:

```
$ dune build && cp _build/default/bin/main.exe main.exe
```

The compliation result of **Poirot** is an executable `_build/default/bin/main.exe`. For the sake of convenience, we copy it under the current directory. You can run **Poirot** by excuting `main.exe <args>` directly, or excuting it via `dune`, that is `dune exec -- bin/main.exe <args>`.

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

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

You can print the refinement type used in the `SizedList` benchmark:

```
$ ./main.exe print-coverage-types meta-config.json data/benchmark/quickchick/sizedlist/_under.ml
```

The expected output is:

```
 Types to Check:
⊢ sized_list_gen : s:{v:int | (0 <= v)}→[v:int list | (∀ u, ((len v u) => ((0 <= u) ∧ (u <= s))))]
```

Coq proofs in the Docker Image

The Coq proofs of our core language λ^{TG} are located in the `coq_proof` directory. These proofs may be executed by running `make`, which may take about `10` min.

```
$ cd coq_proof && make
```

Step-by-Step Instructions

In this section, we provides the instructions to evaluate our artifact. The first half of this section describes installation and use of **Poirot**, an OCaml impelementation of the refinement type checker that verifies the coverage property of the test input generators written in OCaml. The rest of this section describes the Coq formalization of the core language λ^{TG} in the paper and the corresponding soundness theorem.

Artifact Structure

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

- `abstraction/` : the abstract domain built from the method predicates.
- `ast/` and `language/` : the AST of the languages used in **Poirot**.
- `autoverificaiton/` : the Z3 (SMT solver) wrapper.
- `bin/main.ml` : the main entry point of **Poirot**.
- `config/` : the configuration files.
- `coq_proof/` : the Coq proofs of our core language λ^{TG}.
- `doc/` : our submitted papers.
  - `poirot.pdf` : the original submission.
  - `poirot-full-version.pdf` : the original submission with supplemental materials.
- `data/` : the predefined types and the benchmark input files.
  - `data/predefined/` : the predefined types.
  - `data/benchmark/SOURCE/NAME/` : the benchmark input files. The benchmarks are group by thier `SOURCE`. Typically the input source files have name `data/benchmark/SOURCE/NAME/prog.ml`, and the refinement type files have name `data/benchmark/SOURCE/NAME/_under.ml`.
  - The benchmarks of the synthesized results (see more in section Running Benchmarks of Poirot) are saved in the folders that have name with prefix `_synth_`. For example, `data/benchmark/quickchick/_synth_sizedlist/prog.ml` contains all sized list generators that are synthesized by Cobalt.
- `driver` : the IO of **Poirot**.
- `env/` : the univerail environment of **Poirot** which is load from the configuration files.
- `frontend/` : the **Poirot** parser, a modified OCaml parser.
- `meta-config.json` : the main configuration file, the details can be found in Configuration of Poirot.
- `scripts/` : various Python scripts for collecting and displaying experimental results.
- `translate/` : normalization procedure that normalize the code into the Monadic Normal Form (a variant of the A-Normal form).
- `typecheck/` : type check.
  - `typecheck/termcheck.ml` : basic type inference and check.
  - `typecheck/undercheck.ml` : refinement type check.

Running Benchmarks of Poirot

In this section, we discuss the scripts that displayes the tables in the paper.

Comprehensive Scripts

The following scripts run the benchmark suite displayed in Table 1 of the paper, it will take about `50` second:

```
$ python3 scripts/get_table1.py
```

> Notice: in order to solve the new STLC Benchmark, we optimized the implementation of the SMT qeuries (remove unsued quantified variables in the queries), thus the numbers in the `(max. #∀,#∃)` columns would slight different from the table shown in the paper. The execution time `total (avg. time)(s)` may also be various depending on your machine. Readers can check the claims in the paper (line `865` to `880`) with respect to the displayed results.

The following scripts run the benchmark suite displayed in Table 2 of the paper, it will take about `60` mins. It runs Poirot for the programs synthesized using Cobalt deductive synthesis tool.

```
$ python3 scripts/get_table2.py
```

> Notice: the synthesizer uses random choices internally while synthesizing these programs, therefore the programs synthesized in each run of the synthesizer are different and may vary *minutely* in numbers. This may cause *minute* differences in the table generated by the above script from what is reported in the paper. However, in each case, the numbers generated will be close and corroborate our claims in the paper that *the space of the complete generators is significantly smaller than the space of the safe generators.* (line `914`)

The following scripts run the `STLC` benchmark suite that asked by the reviewers, it will take about `200` second. The details about this new benchmarks can be found in section STLC Benchmark.

```
$ python3 scripts/run_stlc.py
```

Detailed Steps

By add command line argument `verbose`, the all scripts above will show the actual command sent to **Poirot** on each benchmark. For example, by runing:

```
$ python3 scripts/get_table1.py verbose
```

The script will print the following commands:

```
Running Poirot on data/benchmark/quickchick/sizedlist/prog.ml
dune exec -- bin/main.exe coverage-type-check meta-config.json data/benchmark/quickchick/sizedlist/prog.ml data/benchmark/quickchick
Running Poirot on data/benchmark/quickchick/sortedlist/prog.ml
dune exec -- bin/main.exe coverage-type-check meta-config.json data/benchmark/quickchick/sortedlist/prog.ml data/benchmark/quickchic
...
```

Readers can try these commands which will take shorter time.

STLC Benchmark

The STLC is a new benchmarks suggested by the reviewers whose setting is not shown in the original version of the paper, thus we provide a details explanation in this section. In this benchmark, **Poirot** will verify the coverage property of a hand-written non-trivial test input generator of the simply typed lambda-calculus (STLC) term written in Coq from the QuickChick. The STLC generator `gen_term_size` will generate well-typed terms of the given type under the given type context with a upper bound of the the number of applications in the term. Besides the main funtion `gen_term_size`, this benchmarks consists of `12` helper functions, which are also traslated from the original implementations (e.g., `gen_term_no_app`, `vars_with_typ`, ...), or the built-in library function in the QuickChick.

The expected expiroment result is shown in the following table. The meaning of the table is the same with the Table 1 in the paper (see "benchmarks" paragraph from line `822`), where we use "▲" indicates these functions are from the STLC benchmark.

	#Branch	#LocalVar	#MP	#Query	(max. #∀,#∃)	total (avg. time) (s)
nonderter_dec ▲	2	4	1	5	(4,2)	0.10(0.02)
gen_const ▲	2	5	2	3	(4,2)	0.05(0.02)
type_eq_size ▲	6†	7	5	9	(10,9)	35.93(3.99)
type_eq ▲	2	6	2	5	(5,2)	0.10(0.02)
gen_type_size ▲	3†	10	1	15	(10,12)	0.48(0.03)
gen_type ▲	2	5	1	3	(4,2)	0.05(0.02)
vars_with_type_size ▲	5†	12	7	13	(12,8)	7.44(0.57)
vars_with_type ▲	2	6	3	6	(5,2)	0.48(0.08)
or_var_in_typectx ▲	3	7	4	4	(5,2)	0.05(0.01)
combine_terms ▲	3	9	6	8	(6,5)	0.14(0.02)
gen_term_no_app_size ▲	3†	13	10	20	(14,15)	4.35(0.22)
gen_term_no_app ▲	2†	6	3	5	(5,2)	0.06(0.01)
gen_term_size ▲	4†	24	9	50	(27,27)	142.93(2.86)

Detail Usage of Poirot

Commands of Poirot

Using **Poirot**, you can

- Print the refinement types (we called *coverage refinement types* in the paper) that encodes the coverage property from the given file:

```
$ ./main.exe print-coverage-types <config_file> <refinement_type_file>
```

> For example,

```
$ ./main.exe print-coverage-types meta-config.json data/benchmark/quickchick/sizedlist/_under.ml
```

> will print

```
Types to Check:
⊢ sized_list_gen : s:{v:int | (0 <= v)}→[v:int list | (∀ u, ((len v u) => ((0 <= u) ∧ (u <= s))))]
```

- Print the source code from the given file in given `format`. Before the refinement type check, **Poirot** (line `811` in the paper) would load the OCaml code (`raw` format), performs the basic type inference (`typed` format), then translate the code in to the Monadic Normal Form (`mnf` format).

```
$ ./main.exe print-source-code [raw | typed | mnf] <config_file> <source_code_file> <refinement_type_file>
```

> For example,

```
$ ./main.exe print-source-code typed meta-config.json data/benchmark/quickchick/sizedlist/prog.ml data/benchmark/quickchick/sizedlis
```

> will print

```
[(Basic) Typed]:

let rec sized_list_gen = (fun (size : int) ->
   (let ((b : bool)) = ((size : int) == (0 : int) : bool) in
    (if (b : bool)
     then ((nil : int list)  : int list)
     else
       ((let ((b1 : bool)) =
           ((bool_gen : unit -> bool) ((tt : unit)  : unit) : bool) in
         (if (b1 : bool)
          then ((nil : int list)  : int list)
          else
            ((let ((size1 : int)) = ((size : int) - (1 : int) : int) in
              (let ((l : int list)) =
                 ((sized_list_gen : int -> int list) (size1 : int) :
                 int list) in
               (let ((n : int)) =
                  ((int_gen : unit -> int) ((tt : unit)  : unit) : int) in
                ((cons : int -> int list -> int list) (n : int)
                   (l : int list) : int list) : int list) : int list)) :
            int list) : int list)) : int list) : int list) : int list) :
int -> int list)
```

- Type check the given source code is type safe with respect to the given refinement type (line `810` in the paper):

```
$ ./main.exe coverage-type-check <config_file> <source_code_file> <refinement_type_file>
```

The result of type check is saved in the file `.result` by default. The first word of `.result` indicates if the code is type safe.

```
< true | false > & <statistics information>
```

> For example,

```
$ ./main.exe coverage-type-check meta-config.json data/benchmark/quickchick/sizedlist/prog.ml data/benchmark/quickchick/sizedlist/_u
```

> The content of `.result` would be:

```
true & sized_list_gen & $4$ & $12$ & $2$ & $11$ & $(7, 9)$ & $0.38(0.03)$
```

You can also turn on the `debug_info.show_typing` in the configuration file (`meta-config.json`) to show the each step of type check. The details about the configuration file is the section Configuration of Poirot.

```
...

Subtyping Check:
size!0:{v:int | (0 <= v)},sized_list_gen:s:{v:int | (0 <= v)}→[v:int list | ((s < size!0) ∧ (s >= 0) ∧ (∀ u, ((len v u) => ((0 <= u)
⊢ [v:int list | (∃ b!14, (∃ x!8, (∃ b!15, (∃ x!9, (∃ b1!7, (∃ a!15, (∃ b!16, (∃ x!10, (∃ b!17, (∃ size1!6, (∃ l!5, (∃ x!11, (∃ n!2,

Task 1, type check succeeded
```

Configuration of Poirot

All commnads of **Poirot** will take a universial configuration file (`meta-config.json`) in JSON format as its first argument. Precisely, the JSON file outputs results in JSON format to some output directory.

- the `debug_info` field controls the debug information output. Precisely, we have the following options:
  - if the `show_preprocess` field is true, **Poirot** will print the preprocess result (line `811` in the paper). It will print the given source code, type code, and the code in Monadic Normal Form.
  - if the `show_typing` field is set as true, **Poirot** will print the type judgement of each step in the type check.
  - if the `show_queries` field is set as true, **Poirot** will print the queries that need to by checked by the SMT solver.
  - if the `show_stat` field is set as true, **Poirot** will print statistics information.
  - if the `show_others` field is set as true, **Poirot** will print any other information (this is for debugging).
- the `resfile` field indicates the path of the output file of type check.
- the `logfile` field indicates the path of the log file of type check.
- the `benchmark_table_file` field indicates the path of benchmarks.
- the `prim_path` field indicates the predefined coverage types for a number of OCaml primitives, including constants, various arithmetic operators, and data constructors for a range of datatypes(line `813` to `815` in the paper).

Input File Formats

The source code file expected by **Poirot** is simply a OCaml functions listing. Currently, **Poirot** handles only a subset of OCaml, it does not handle features involving references and effects, parametric polymorphism, or concurrency. Additionally, all functions should be annotated with precise input and output type; all left-hand-side variable in the let binding should be annotated with its precise type.

The refinement type file expected by **Poirot** is also a OCaml source code file with specially formatted:

```
(* The method predicates used in the current benchmark *)
external method_predicates : t = "NAME" "NAME" "NAME" ...

(* The refinement type of the library function NAME, which doesn't need to be type check *)
let[@library] NAME = UNDER_APPR_RTY

...

(* The refinement type of the function NAME, which needs to be type check *)
let NAME = UNDER_APPR_RTY

...
```

where `NAME` is simply a string.

The method predicates are predefined uninterrupted functions that capture non-trivial datatype shape properties (see line `815` to `821` in the paper). The semantics of the method predicates are define in the file `data/predefined/axioms_of_predicates.ml` .

The syntax of the `UNDER_APPR_RTY` (underapproximate refinement type, a synonyms of coverage type) is similar with the OCaml let expression but with attributes:

```
 VAR := string
OCAML_TYPE:= the OCmal type

METHOD_PREDICATE := the method predicate
OP := "==" | "!=" | "+" | "-" | "<" | ">"

TYPED_QUANTIFIER := "[%forall: OCAML_TYPE]" | "[%exists: OCAML_TYPE]"

LIT :=
| "true"
| "false"
| VAR
| "VAR OP VAR"

PROP :=
| LIT
| "METHOD_PREDICATE LIT ..."
| "implies PROP PROP"
| "iff PROP PROP"
| "PROP && PROP"
| "PROP || PROP"
| "not PROP"
| "fun (VAR : TYPED_QUANTIFIER) -> PROP"

UNDER_APPR_BASE_RTY := "(PROP : [%VAR: OCAML_TYPE]) [@under]"

OVER_APPR_BASE_RTY := "(PROP : [%VAR: OCAML_TYPE]) [@over]"

UNDER_APPR_RTY :=
| UNDER_APPR_BASE_RTY
| "let VAR = OVER_APPR_BASE_RTY in UNDER_APPR_RTY"
| "let VAR = UNDER_APPR_BASE_RTY in UNDER_APPR_RTY"
```

where the `METHOD_PREDICATE` is the method predcicates introduced in the first line of the file; the quantifers in the first-order logic (FOL) proposition `PROP` is typed (`TYPED_QUANTIFIER`).

Currently, the `OCAML_TYPE` supported by the **Poirot** is fixed, which is defined in the file `data/predefined/data_type_decls.ml` .

The defintion of the coverage type is consistent of the Figure 3 (line `359`), which consists of both "overapproximated-style" refinement type and the "overapproximated-style" refinement type. Precisely,

- the overapproximate refinement type `{v:b | ϕ}` in the paper is defined as `OVER_APPR_BASE_RTY` .
- the underapproximate refinement type `[v:b | ϕ]` in the paper is defined as `UNDER_APPR_BASE_RTY` .
- the function type is defined as let expression. We use the let binding to represent the argument type and use the body of the let expression to represent the return type. For example, `let x = t_x in t` represents the type `x:t_x→t` . Here we syntactically disallow the underapproximate base refinement type to be the agrument type following the constraints in the paper (line `422` to `426`).

Proof Readme of λ^{TG}

Proof File Structures

The files are structured as follows:

- Definitions and proofs of our core language λ^{TG} that are independent of **Poirot**'s type system.
  - `Atom.v` : Definitions and notations of atoms (variables) in λ^{TG}.
  - `Tactics.v` : Some auxiliary tactics.
  - `CoreLang.v` : Definitions and notations of λ^{TG}.
  - `NamelessTactics.v` : Auxiliary tactics for the locally nameless representation.
  - `CoreLangProp.v` : Lemmas for our core language λ^{TG}.
  - `OperationalSemantics.v` : Definitions and notations of the small-step operational semantics of λ^{TG}.
  - `OperationalSemanticsProp.v` : Lemmas for the small-step operational semantics of λ^{TG}.
  - `ListCtx.v` : Definitions and notations for reasoning about polymorphic contexts.
  - `BasicTyping.v` : Definitions and notations for the basic typing.
  - `BasicTypingProp.v` : Lemmas for the basic typing rules.
  - `SyntaxSugar.v` : Definitions and notations of the syntax sugar.
  - `TermOrdering.v` : Auxiliary definitions and notations of the partial order relation over terms.
  - `RefinementType.v` : Definitions and notations of coverage types.
  - `RefinementTypeTac.v` : Auxiliary tactics for coverage types.
  - `RefinementTypeDenotation.v` : Definitions and notations of the type denotation (without type context).

- `RefinementTypeDenotationTac.v` : Auxiliary tactics for the type denotation (without type context).
- `RefinementTypeDenotationProp.v` : Lemmas for the type denotation (without type context).
- `WFCtxDenotation.v` : Auxiliary definitions and notations of the denotation of well-formed type context.
- `WFCtxDenotationProp.v` : Lemmas for the denotation of well-formed type context.
- `InvDenotation.v` : Auxiliary definitions and notations of the invariant denotation.
- `InvDenotationProp1.v` ... `InvDenotationProp4.v` : Lemmas fpr the invariant denotation.
- `CtxDenotation.v` : Definitions and notations of the type denotation under type context.
- `CtxDenotationProp.v` : Lemmas for the type denotation under type context.
- `Typing.v` : Definitions and notations used in the typing rules.
- `Soundness.v` : Statement and proof of the soundness theorem.

Compilation and Dependencies

The Coq proofs of our core language λ^{TG} are located in the `coq_proof` directory.

```
$ cd coq_proof
```

The project is organized by the coq make file `_CoqProject`, which may be executed by running `make` (about `10` min).

```
$ make
```

Our formalization is tested against `Coq 8.14.1`. We also rely on the library `coq-stdpp 1.7.0`.

Our formalization takes inspiration and ideas from the following work, though does not directly depend on them:

- Software Foundations: a lot of our formalization is inspired by the style used in Software Foundations.
- The Locally Nameless Representation: we use locally nameless representation for variable bindings.

Paper-to-artifact Correspondence

Definition/Theorems	Paper	Definition	Notation
Syntax	Figure 3	mutual recursive defined as `value` and `tm` in file `CoreLang.v` (line `42`)	
Operational semantics	Figure 11 (supplementary materials)	`step` in file `OperationalSemantics.v` (line `23`)	`e ↪ e'`
Basic typing rules	Figure 12 (supplementary materials)	mutual recursive definition of `tm_has_type` and `value_hsa_type` in file `BasicTyping.v` (line `56`)	`Γ ⊢t e ⋮t T` and `Γ ⊢t v ⋮v T`
Well-formedness typing rules	Figure 4	`wf_typing` in file `Typing` (line `44`)	`Γ ⊢WF τ`
Subtyping rules	Figure 4	`subtyping` in file `Typing` (line `97`)	`Γ ⊢ τ1 <: τ2`
Disjunction typing rules	Figure 4	`disjunction` in file `Typing` (line `92`)	`Γ ⊢ τ1 <: τ2 <: τ3`
Selected typing rules	Figure 5 (Figure 13 in supplementary materials shows full set of rules)	given by the mutually inductive types `value_under_type_check` and `term_under_type_check` in file `Typing.v` (line `149`)	`Γ ⊢ e ⋮t τ` and `Γ ⊢ v ⋮v τ`
Type denotation	At the beginning of Section 4.1 (line `573` to `576`)	`rR` in file `RefinementTypeDenotation.v` (line `23`)	`{ n; bst; st } ⟦ τ ⟧`
Type denotation under context	At the beginning of page 13 (line `594` to `597`)	`ctxrR` in file `CtxDenotation.v` (line `25`)	`{ st }⟦ τ ⟧{ Γ }`
Soundness theorem	Theorem 4.2	`soundness` in file `Soundness.v` (line `140`)	

> Readers can find the supplemental materials in `doc/poirot-full-version.pdf`.

Differences Between Paper and Artifact

- Our formalization only has two base types: nat and bool.
- To simplify the syntax, our formalization don't treat the operators (e.g. `+`) as value (line `349` in the paper). Alternately, we can define the operators as value using lambda functions. For example, the value `+` can be defined as

```
fun (x: nat) (y: nat) =
    let (res: nat) = x + y in
    res
```

- Our formalization only have four operators: `+`, `==`, `<`, `nat_gen`. Other operators shown in the paper can be implemented in terms of these. E.g., the minus operator can be defined as:

```
let minus (x: nat) (y: nat) =
    let (diff: nat) = nat_gen () in
    if x + diff == y then diff else err
```

In addition, to simplify the syntax, all operators take two input arguments; e.g., the random nat generator takes two arbitrary numbers as input.

- In the formalization, to simplify the syntax, pattern-matching can only pattern match against Boolean variables. Pattern matching over natural numbers

```
match n with
| 0 -> e1
| S m -> e2
```

is implemented as follows:

```
if n == 0 then e1
else let m = n - 1 in e2
```

- We assume all input programs are alpha-converted, such that all variables have unique names.
- We use the locally nameless representation in all terms, values, refinement, and type context, thus the definition looks slight different from the definition shown in the paper.
- We encode the propositions in the refinement type as Coq propositions. In order to capture the the free variables and the bound variables (see locally nameless repsentation), all proposition will constructed with
  - a set of atoms (variables) `d`, which is the of the free variables in the proposition.
  - a natrual number `n`, which indicate the upper bound of the bound variables.
- Following the encoding above, for example, the base coverage type `[v:b | ϕ]` will be encoded as `[v:b | n | d | ϕ]`.
- The substitution of refinement types are formalized into states (a mapping from varaibles to values), helping to eliminate termination checks of the fixpoint function in Coq when we define the logical relation. Precisely
  - the definition of the type denotation has the form `{ n; bst; st }⟦ τ ⟧` instead of `⟦ τ ⟧`, where `(n, bst)` is the state of the bound variables, `st` is the state of the free variables.
  - the definition of the type denotation under the type context has the form `{ st }⟦ τ ⟧{ Γ }` instead of `⟦ τ ⟧{ Γ }`, where `st` is the state of the free variables. Here we ommit the bound state `(n, bst)` thus all types in the type context are locally closed (see locally nameless repsentation).
- In the formalization, our coverage typing rules additionally require that the all branches of a pattern matching expression are type safe in the basic type system (they may not be consistent with the coverage type we want to check). We didn't mentioned it in the original paper, however, we will fix it in the second round submission.
- To the sake of convenience of proof, we split a single typing rules into different cases (then we can prove these cases separately dyring the induction proof). For exmaple, the rule `TLetE` in the figure 13 (in the supplementary materials) is split as `UT_Lete_base` and `UT_Lete_arr` in our proof.

Appendix: Building Cobalt from Source (Optional)

Cobalt Synthesizer

Cobalt (based on the paper) is a purely functional, refinement-type guided synthesizer. This takes a pure-function spec, along with a small library (function signatures) and synthesis all possible programs of a given function call length `k` and given nested depth. Unfortunately, the depedencies (e.g., the version of the OCaml and z3) of `Cobalt` is conflit with **Poirot**, thus we put the synthesized results (see `data/` feild in the section Artifact Structure) instead of the synthesizer into the docker.

This step is optional: although the program synthesis is not the claims in the our paper, we still provide instructions to reproduce these synthesized programs for the readers who have a deep interest in it. In the rest of this section, 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

Running Cobalt:

Cobalt takes the following arguments:

```
$ ./effsynth.native [-cdcl] [-bi] [-k] [-nested] <path_to_specfile>

$ ./effsynth.native -cdcl -bi -k 3  tests_specsynth/ulist_quant.spec
```

This should produce a list of synthesized programs in `output/tests_specsynth/ulist_quant.spec` directory.

Generating Poirot benchmarks:

The 5 benchmarks in `Table2` in **Poirot** are in `test_specsynth/Poirot_benchmaks` directory.

Run the following command to generate the programs used in Poirot for UniqueList:

```
$ ./effsynth.native -cdcl -bi -k 3 tests_specsynth/Poirot_benchmarks/Poirot_uniquelist.spc
```

This will generate a file `output/tests_specsynth/Poirot_benchmaks/Poirot_uniquelist.spc` containing the required programs.