CMPT489 PROJECT PHASE 1 REPORT

JOSEPH LUNDERVILLE (200014381)

FALL 2023 – YUEPENG WANG

DESIGN

The basic design is straightforward: a local class `TopDownEnumSynthesizer.EnumerationJob` encapsulates a partial AST. Its main method `enumerate(validate, jobs)` enumerates programs by substituting all possible productions into its first available missing AST node; if this results in a complete AST, it is submitted to the `validate` function supplied to it, otherwise it creates a new `EnumerationJob` for the partial AST and enqueues it in the `jobs` queue supplied.

The caller is expected to iteratively remove jobs from the queue and call their `enumerate(...)` methods.

FEATURES

The program generates extra output in exceptional situations; when it hits a limit of partial programs queued, it outputs the message:

`warning: enumeration capped at <N> pending jobs`

and when it is unable to synthesize a program matching the examples, it outputs the message:

`error: failed to generate!`

TESTS

Besides the supplied test case, I implemented a Haskell program (`test-data/gen/src/Main.hs`) which generates random programs in the supplied DSL, and evaluates them with random inputs to create a test case. These programs are generated with different maximum AST depth to produce test cases of varying challenge.

This method is imperfect in a number of ways; mainly, the inputs may not distinguish the generated program effectively from other potential programs; and even if the inputs do effectively distinguish the generated program, it is possible the generated program is entirely equivalent to another program with lower depth.

As an example of the first, take the program **`Ite(And(Lt(x, y), Lt(x, z)), 2, y)`** the generator might produce input-output examples:

```
x=2, y=3, z=2 -> 3
x=1, y=2, z=3 -> 2
x=2, y=3, z=1 -> 3
x=1, y=1, z=3 -> 1
```

Despite actually resulting in 100% code coverage, these examples are also consistent with the trivial program **`y`**.

More problematic is the second situation, where the program may be entirely equivalent to another, simpler program. For example, consider the program **`Ite(And(Lt(x, y), Lt(y, x)), 2, y)`**: because $x<y \wedge y<x$ is always false, we can show that the program is equivalent to the trivial program **`y`**. However, this is not practical to detect during test case generation.

In both cases the solution was to generate an abundance of test cases and spot pathological ones manually. Essentially, if a more trivial than expected program is synthesized, I manually checked if it was equivalent and replaced the test case with a freshly generated one. This obviously only works for programs where a solution was synthesized, but the problem of test cases which are too difficult for the synthesizer but still not quite as challenging as I would like, is a problem which can be solved progressively as the synthesizer is improved.

ISSUES

The enumeration order is suspect: the synthesis sometimes produces deeper programs before shallower ones. I produced a second variation which is uses progressively deeper depth-first searches to reduce memory consumption, and this one generates programs in the correct order, but it is significantly slower and not able to find any programs with AST height greater than 3 either in any reasonable time. Due to time constraints I’m abitrarily choosing to submit the faster one without fully exploring its behaviour.