CSC410

Project: Enumerative Synthesizer

Due on Wednesday December 8, 2021 at 11:59pm

Project Format and Guidelines on Submission

This project is worth 30% of the total course mark. Submit on Markus and follow these rules:

- This is a group project, designed for groups of 3 students. You can use Piazza, Quercus, or any means you see fit to form groups. After the drop date, we can accommodate stranded students by relaxing this requirement to make sure they find a host group.
- Each group should submit an archive called `project.zip`. Note that Markus has been set up to accept exactly this file.
- Download the starter code from Github. The repository contains all you need to get started. For the software requirements, please refer to the `README.md` file at the root.
- The starter code includes a number of files. You should at least modify the files `main.py`, `lang/symb_eval.py`, `synthesis/synth.py`, `verification/verifier.py` and `test/student_test.py`. You are welcome to add new files (and new functions, new classes) but you should not change the name of the existing classes and functions, as well as the names of the existing files.

The submission system will remain open for 12 hours after the deadline, but there is a penalty deduction formula set in Markus that deducts 4% for every hour of late submission up to 12 hours.

It is recommended that you read this whole document before starting work on this project. In particular, read the section *Style, Testing, and Documentation*, since you should be using the correct coding style and writing tests and documentation throughout the term – not just at the very end.

1 Problem Definition

1.1 Overview

In this project, you build a program synthesizer, that is, a tool that takes as input a program with holes, and completes the holes in that program such that the completed program is correct, according to some condition.

Your synthesizer works in the following manner: it guesses an implementation for the holes and then evaluates the program symbolically with the chosen implementation. The result is a symbolic expression that needs to be verified. If that expression is a valid formula, then the conjectured implementation (for the holes) is a (valid) solution. Otherwise, another conjecture is made, and this process continues until a solution is found.

1.2 Input Language Syntax

The programs for this project will be written in a toy language, called `Paddle`, which supports only two types of variables: booleans and integers. Additionally, `Paddle` supports the means of providing a specification for the synthesis problem.

Each problem instance (in `Paddle`) consists of four separate components, that appear in the following order in the file:

1. A of list **input variable declarations**. An input variable declaration has the following syntax:

   `input <identifier> : <type> ;`

   where `<identifier>` is a string and `<type>` is `int` or `bool`.

2. A set of **hole declarations** with the syntax:

   `hole <identifier> : <type> [ <grammar> ] ;`

   where `<identifier>` is a string and `<type>` is `int` or `bool`, and you will find a definition for grammars below.

3. A set of **assignments** of the form:

   `define <identifier> : <type> = <expression> ;`

   where `<identifier>` is a string and `<type>` is `int` or `bool`, and the syntax for the `<expression>` is given in Appendix A.1.

4. A **correctness constraint**

   `assert <expression> ;`

Note that, in `Paddle`, a variable can only be declared once, either as an input, a hole, or in an assignment.

Example 1.1. Here is a valid `Paddle` instance:

```
input x : int;
input y : int;
hole hmax : int [ G : int -> G + G | B ? G : G | Var | 0 | 1;
                  B : bool -> G > G | G < G | G = G | B && B | B || B | ! B];
define c : int = hmax;
assert ( c >= x && c >=y && (c = x || c = y));
```

First, it defines the input variables `x` and `y`. Then, it defines a hole `hmax` with a grammar. The statement after the hole definition is an assignment: the variable `c` is defined and assigned the value of the hole (which is unknown for now). Finally, the last statement must be the correctness condition. In this example, it corresponds to asserting that `c` is the maximum of `x` and `y`.

1.2.1 Grammars

A `<grammar>` defines a universe (set) of expressions. It is a list of production rules. Each production rule is of the following form:

`<identifier> : <type> -> <grammar_expression> | <grammar_expression> | ...`

and the list of the rules is entered as a `;`-separated list

Each production rule has a non-terminal (the identifier with its type) on the left of `->` and a list of productions separated by `|` on its right. The `<grammar_expression>` is either an `<expression>`, or the keyword `Var`, or the keyword `Integer`. An `<expression>` in this context is restricted to only use non-terminals as variables. `Var` can be replaced by any variable that is declared before the first appearance of the hole (that this grammar is associated with) in the assignments. `Integer` stands for any integer-typed constant.

Example 1.2. In Example 1.1, we defined the following hole:

```
hole hmax : int [ G : int -> G + G | B ? G : G | Var | 0 | 1;
                  B : bool -> G > G | G < G | G = G | B && B | B || B | ! B];
```

A completion for hole `hmax` is a member of the language defined by the grammar. For example, 0 is a possible completion since it is one of the productions for the rule associated to `G`. We can also replace `hmax` by `x > y ? x : y` (the ternary C expression, equivalent to "if `x < y` then `x` else `y`"). This expression is produced by taking the production `B ? G : G` in the rule for `G`, then replacing `B` by one of its productions `G > G`. Then the occurrences of `G` can be replaced by `Var` and `Var` can be replaced by `x` or `y`, since those are the two variables defined before `hmax` is used in the assignment.

The `examples` folder in the starter code contains a large set of examples.

1.3 Semantics

A `Paddle` file defines a synthesis problem. Here we formally define what a *valid solution* to a synthesis problem is.

- Let $I = \{i_1, \ldots, i_n\}, n \geq 0$ be the sets of input variables (the variables defined by an `input` statement).
- Let h be a set of hole completions, i.e. a map from hole ids to an expression that belongs to the hole grammar.
- Let $e(h)$ be the result of symbolically evaluating the program (see Section 2) with the hole completions h. Remark that the only variables appearing in $e(h)$ are the variables in I.

The program is *correct* given the hole completions h iff the formula $\forall i_1, \ldots, i_n.e(h)$ is valid. The hole completion h is a *valid solution* to a synthesis problem if and only if the program is *correct* given h.

Example 1.3 (Example 1.1, continued.)**.** In our example, among the infinite possibilities we have to complete the holes, we can start with the smallest one: `hmax = 0` Let us check whether the program is correct with this hole completion. We first evaluate the program. The assignment `c = hmax` translates to `c = 0`. Then we replace c by its expression in the correctness constraint, which becomes:

```
0 >= x && 0 >= y && (0 = x || 0 = y).
```

The formula $\forall x, y.0 \geq x \wedge 0 \geq y \wedge (0 = x \vee 0 = y)$ is obviously invalid!

Another possible completion for `hmax` is `hmax = x > y ? x : y`. This is a correct solution (you can try evaluating the program manually, and convince yourself that the correctness constraint is valid).

2 Symbolic Evaluation (2 points)

As the first step in this project, you will write a function that symbolically evaluates the program. We have provided a partial implementation in `lang/symb_eval.py`.

Symbolically evaluating a program with holes consists in executing each assignment and then replacing the variables in the constraint by their computed expression, and the holes by their completions. The result of the symbolic evaluation is an expression where the only variables allowed are the input variables.

Example 2.1. The following program has one hole and should compute the sum of three integers. Let us assume that we have the hole completion $\{h \rightarrow z\}$.

```
input x ; int;
input y : int;
input z : int;
hole h : int [ G: int -> G + G | Var ]; // completion: h -> z
define a : int = x + y; // h-> z, a -> x + y
define b : int = a + h; // h -> z, a -> x + y, b -> x + y + z
assert (b = x + y + z);
```

The result of symbolically evaluating the program is the expression $(x + y + z) = (x + y + z)$. In the first assignment, a is assigned $x + y$ (there is no expression to replace). In the second assignment, the a in the expression $a + h$ is replaced by its (symbolic) value, $x + y$. Then, the hole h is replaced by its completion z. Then, b is assigned $x + y + z$. Finally, b is replaced by its (symbolic) value in the correctness constraint, which yields the expression $(x + y + z) = (x + y + z)$.

The goal of this part of the project is for you to understand the internal representation of a `Paddle` program (in the file `lang/ast.py`), and also to get a more precise understanding of the general problem. You should spend some time reading the test file `test/eval_test.py` and the file you need to complete `lang/symb_eval.py`.

Complete the parts indicated with a `#TODO..` in the function `evaluate_expr` of the file `lang/symb_eval.py`. The `evaluate_expr` method is part of an `Evaluator` class, which is initialized with the definitions for the holes. The `evaluate_expr` takes two arguments: (i) a mapping from variables to expressions (the symbolic values of the variables) and (ii) the expression that needs to be evaluated. Carefully read the comments in the file as they are meant to help you. The amount of code you need to write is minimal here; at most two lines per `#TODO` comment. Once you are done, you should be able to uncomment the line corresponding to this part in `test.py`:

```
# TODO Once you have completed 2 - Evaluating Expression, uncomment the next line!
# from test.eval_test import *
```

After uncommenting, executing `python test.py` should not return any errors.

3 Verifying Programs (9 points)

In the context of your synthesizer, the only programs you need to verify are `Paddle` expressions. Write the implementation of the function `is_valid` in `verification/verifier.py`. This function takes as input an instance of an `Expression` (from `lang/ast.py`) and returns a boolean. Once you are done, you can uncomment the corresponding tests in `test.py`:

```
# TODO Once you have completed 3 - Verifying Programs, uncomment the next line
#from test.verif_test import *
```

Running the tests should not return any errors.

4 Enumerating Programs (10 points)

Your synthesizer proceeds by enumerating all possible hole completions and then verifying them. You will write three different completion strategies in the `Synthesizer` class of the file `synthesis/synth.py`: `synth_method_1`, `synth_method_2`, and `synth_method_3`. For a given instance `s` of a `Synthesizer`, a new set of hole completions should be returned each time a call to one of the synthesis methods is made. Please read `synthesis/synth.py` carefully.

Write the implementation of the functions in `synthesize/synth.py`. Once you are done, you can uncomment the corresponding tests in `test.py`:

```
# TODO Once you have completed 4 - Symbolic Evaluation, uncomment the next line
# from test.enumerate_test import *
```

Running the tests should not return any errors. Note that the tests in this file are basic tests that check that the implementations returned by your functions satisfy some basic properties. It does not check that the synthesized programs are correct, and therefore you can start implementing this part without implementing Part 3. **Once you have completed Part 3 and Part 4**, you can try synthesizing some solutions. You should also check that all tests pass when uncommenting the following:

```
# TODO Once you have completed 3 and 4, uncomment the next line.
# from test.synth_test import *
```

5 Style, Testing, and Documentation (9 points)

Style (1 points)

Although you are not required to follow a specific coding style guide, your code should be readable and well-commented. Do not leave TODOs, unused variables and functions in your final submission. The following command should return `0` when executed at the root of the project:

```
$ flake8 . --count --exit-zero --max-line-length=127 --ignore F,W503,W504,E722
```

Testing (6 points)

You must write tests for the various components of this project - at least 5 for each part, so 5x3 = 15 tests. This project uses a testing framework called `unittest`. You may add your tests to the file `test/student_test.py`. Your tests should be methods of the class `TestStudent` and their names should be prefixed with `test_`. You should avail yourself of the methods provided by the `unittest` framework, such as `assertTrue`, `assertEqual`, and `assertRaises`. Here are some useful commands:

```
$ python3 ./test.py # run all tests that are imported in test.py
$ python3 ./test.py -v # verbose output
$ python3 ./test.py TestStudent # run all tests in the TestStudent class
$ python3 ./test.py TestStudent.test_sanity_student # run one test
$ python3 ./test.py -k test_verif # run all tests with names containg substring "test_verif"
```

Documentation (2 points)

The starter code has been set up so that html documentation is generated when you execute the command `pycco *.py **/*.py` at the root. You can open `docs/main.html` in your browser. You should document every function you write. We also expect the following pieces of documentation:

1. The names and UTORids of each of your team members, in `main.py`.
2. A very brief description of what each team member did in the project, in `main.py`.
3. A description of your tests, for each test you write in `test/student_test.py`.
4. A description of your synthesis strategies in the documentation of `synth_method_1`, `synth_method_2`, and `synth_method_3` in `synthesize/synth.py`.

6 Bonus (up to 4 points)

Performance of method 3 (2 points). Each group's `synth_method_3` will be timed to evaluate its performance on a set of synthesis benchmarks. Students whose times exceed a "better than naive" threshold designed by the TA team will receive 1 bonus point. The fastest group overall will receive one additional bonus point. Therefore, in total 2 bonus points are up for grabs in the performance category.

Analysis of the performance of three methods (2 points). Your analysis should include (at least) a comparison of the success rate and the timings of the three methods on a set of benchmarks of your choosing. You should present your quantitative results in a table and/or graph, and include a thoughtful analysis of your results. You should write your analysis in a report named `evaluation.pdf` and include it in your zip file.

A Appendix

A.1 Program Syntax

The program syntax:

```
// A program is a list of <input_decl>, <hole_decl>, <assignment> and <assert>,
// precisely in that order.
// There should be exactly one <assert>
<program> := <input_decl>* <hole_decl>* <statement>* <assert>
// Input variable declaration
<input_decl> := input <identifier> : <type>;
// Type is bool or int
<type> := bool | int
// Hole Declaration
<hole_decl> := hole <identifier> : <type> [ <grammar> ];
//Assignment assigning value to fresh variable
<assignment> := define <identifier> : <type> = <expression>;
// The correctness constraint is just a boolean expression
<assert> := assert <expression>;
```

Where an `<expression>` has the following syntax:

```
<expression> ::=
  | <identifier> // the id of a variable
  | Integer  // An integer constant
  | True | False
  | ( <expression> ) // an expression in parentheses
  | <expression> ? <expression> : <expression> // an if-then-else expression
  | <expression> + <expression>
  | <expression> - <expression>
  | <expression> / <expression>
  | <expression> % <expression>
  | abs <expression>
  | - <expression>
  | <expression> = <expression>
  | <expression> != <expression>
  | <expression> < <expression>
  | <expression> <= <expression>
  | <expression> >= <expression>
  | <expression> > <expression>
  | <expression> && <expression>
  | <expression> || <expression>
  | ! <expression>
```

And a `<grammar>`has the following syntax:

```
// A grammar has at least one production rule. The first production rule is the
// "main" rule of the grammar and uses the top-level symbol.
<grammar> := <production_rule> | <production_rule> ; <grammar>
// A production rule maps an Id to a Production
<production_rule> := <identifier> : <type> -> <production>
// A production is an list of expressions separated by ‘|‘
<production> := <grammar_expression> | <grammar_expression> | <production>
```

```
// A grammar expression is an Expression or the special keywords Var and Constant
// Integer stands for any integer, Var stands for any variable
<grammar_expression> := <expression> | Var | Integer
```