EECS 598 and EECS 498 (Fall 2021) – Assignment 2
Due date: Monday, September 27 at midnight EST

Collaboration. You are free to discuss the assignment with others or work together towards the solution. However, all of your code must be written by yourself. Your write-up must be your own as well. **Please list your collaborators at the top of your submissions.**

Goal. The goal of this assignment is to help you get familiar with `z3` as well as how to use z3 for pruning. In particular, in this assignment, you will first use `z3` to solve a simple SMT solving problem and then use `z3` in our program synthesizer to perform deduction-based pruning.

1 Getting started

Download `A2` from
https://web.eecs.umich.edu/~xwangsd/courses/f21/assignments/pa2.zip
which contains a `pa2` folder. Put it under `syn`.

```
codebase
├── src
│   └── main
│       └── java
│           └── syn
│               ├── base
│               ├── pa0
│               ├── pa1
│               └── pa2
│                   ├── Synthesizer2.java
│                   ├── Test4.java
│                   ├── Test5.java
│                   ├── Test6.java
│                   └── Test7.java
└── lib
    ├── com.microsoft.z3.jar
    ├── libz3.dylib
    └── libz3java.dylib
```

First of all, `Synthesizer2` extends `Synthesizer1` and provides an implementation for `run`. You will use this implementation in this assignment. Furthermore, `Synthesizer2` also includes two additional functions, `attemptToPrune` and `prune`, which are two important functions you will complete in this assignment. There is also a `createSpecs` function whose implementation is already given. That means, you don't need to create the specs yourself but you will need to use them in this assignment.

In this assignment, we will also use the Microsoft `z3` solver. Necessary libraries are already included in the `lib` folder but you will need to add this folder to your `DYLD_LIBRARY_PATH`, for example, by adding the following line in `.bash_profile` if you use MacOS.

```
export DYLD_LIBRARY_PATH=path_to_lib:${DYLD_LIBRARY_PATH}
```

To test whether you set it up correctly, run the `main` function in `Test4`. If things are all set up correctly,

you should see something like: "(= x 1) is SATISFIABLE". It's possible you will need to install `z3` from scratch yourself. The source can be downloaded from here: https://github.com/Z3Prover/z3. Note that you will need Java bindings when you compile it.

It's possible you'll need to build `z3` from its source. You can find some instructions on how to do this from here: https://github.com/Z3Prover/z3. Generally speaking, you will need to go through the following steps.

- Get the source. You can choose to clone the `z3` repository. Or you may download a `z3` .zip file from here: https://github.com/Z3Prover/z3/archive/refs/tags/z3-4.8.12.zip.
- Run `python scripts/mk_make.py --java`. Note that you have to include `--java` in order to build `z3` with the Java binding.
- Go to `build` directory, run `make` and then `sudo make install`. This will install `z3` locally on your machine. Alternatively, you can run `make examples` which does everything and also tests the install on some test cases.
- Set the library path. If you use Linux, you should set `LD_LIBRARY_PATH` to the `build` directory. If you use MacOS, you need to set `DYLD_LIBRARY_PATH` to the `build` directory.
- You should see a `com.microsoft.z3.jar` file in your `build` directory. Move it to the `lib` folder.
- Now you can run `Test4`. For example, you can directly run `Test4` from Visual Studio Code, since the class path is already specified in `pom.xml`. If you use command line, you will need to first compile the project: `mvn compile`. Then, you can run `Test4`:

  ```
  java -cp lib/com.microsoft.z3.jar:target/classes syn.pa2.Test4.
  ```

2 Get familiar with `z3`

`Z3` is a cross-platform satisfiability modulo theories (SMT) solver that was developed by Microsoft Research and is still under active development. It is open-source: https://github.com/Z3Prover/z3. There ~~is~~ was an online playground:
https://rise4fun.com/Z3/Aqz2
with a tutorial:
https://rise4fun.com/Z3/tutorial/guide.
However, these two websites seem to be down recently. We're not quite sure if they will be up again in the future, but you can use https://compsys-tools.ens-lyon.fr/z3/index.php which seems to work quite well. We will use the `z3` Java binding for our assignments. Here are some examples about how to use the Java API: https://github.com/Z3Prover/z3/tree/master/examples/java .

In this assignment, use `z3` to create the following formula and check its satisfiability: $x > y \wedge x = 2 \wedge y = 3$**.** Write your code in `Test5.java`. You are highly **encouraged** to try more `z3` examples to get more hands-on experience but we do not require you to submit those examples in this assignment.

3 Prune search space

Now you have some hands-on experience with `z3`. Next, we will use `z3` in our program synthesizer. As you have already witnessed in Assignment 1, `Synthesizer1` was very slow, because it essentially enumerates *all* programs in the CFG and checks them one by one against the example. This is not going to work well for

large CFGs. In this assignment, we are going to use `z3` to perform *automated logical deduction* and prune *partial* programs. That is, we are going to encode a partial AST with holes into a `z3` constraint ϕ. The idea is, if ϕ is that unsatisfiable, that means there is no way this partial AST would lead to a correct program that satisfies the example. In what follows, we will explain how to do this concretely.

We are still going to work with the example used in Assignment 1.

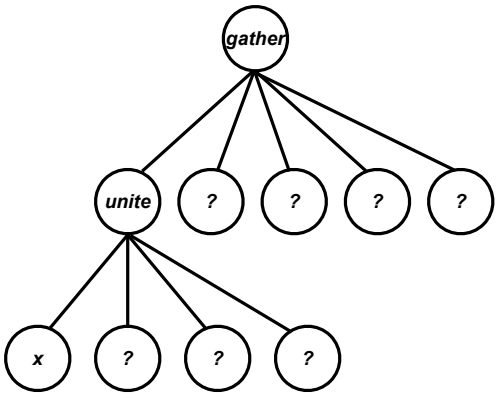
Input (4×4 table):

	var	val	round	nam
1	22	0.1	round1	foo
2	11	0.2	round2	foo
3	22	0.5	round1	bar
4	11	0.9	round2	bar

Output (2×5 table):

	nam	val_round1	val_round2	var_round1	var_round2
1	bar	0.5	0.9	22	11
2	foo	0.1	0.2	22	11

Consider the following AST that represents a partial program P: *gather*(*unite*(x, ?, ?, ?), ?, ?, ?, ?).



In Assignment 1, our `Synthesizer1` would generate this partial program P at some point, it would add P into the worklist, and the algorithm would keep enumerating *all* ASTs that can be expanded from P. However, we know that this P wouldn't lead to a correct program that satisfies the given example, because neither of the operators *gather* and *unite* would increase the number of columns in the table. Since the input example has fewer columns than those in the output example, it is not possible for any expansion of this P to satisfy the given example, no matter what concrete parameters we choose for the ?'s.

We are going to use `z3` to perform the aforementioned reasoning *automatically* to prune out this partial program P. First, we would need a specification for each operator.

Operator	Specification
gather	$x_{out} \leq x_{in} \wedge y_{out} \geq y_{in}$
unite	$x_{out} = x_{in} - 1 \wedge y_{out} = y_{in}$

Here, looking at the specification for *gather*: x_{in} denotes the number of columns of the input table for *gather* function, x_{out} is the number of columns of the table returned by *gather*. Similarly, y_{in} and y_{out} correspond

to the number of rows in the input and output of *gather* respectively. Note that this specification is consistent with the actual semantics of *gather*. That is, this specification *over-approximates* the actual implementation of *gather*. The specification for *unite* is very similar.

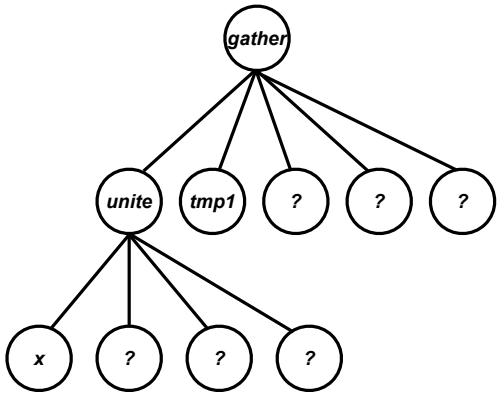
Now, given specifications of *individual operators*, we will construct a "bigger" constraint ϕ for a *partial program*. For example, the constraint ϕ for the previous partial program P is shown as follows.

$$\begin{array}{ll} \phi = \phi_1 \wedge \phi_2 \wedge \phi_3 \wedge \phi_4 & \\ \phi_1 = (x^0_{out} = 4) \wedge (y^0_{out} = 4) & \phi_2 = (x^1_{out} = x^0_{out} - 1) \wedge (y^1_{out} = y^0_{out}) \\ \phi_3 = (x^2_{out} \leq x^1_{out}) \wedge (y^2_{out} \geq y^1_{out}) & \phi_4 = (x^2_{out} = 5) \wedge (y^2_{out} = 2) \end{array}$$

Here, ϕ is a conjunction of four constraints $\phi_1, \cdots, \phi_4$. ϕ_1 corresponds to the input example which encodes the fact that the input has 4 columns and 4 rows. ϕ_2 corresponds to the root node, *gather*, of P. Similarly, ϕ_3 corresponds to *unite*'s specification. Note that we need to properly rename the specifications in order to avoid name collision. Finally, ϕ_4 corresponds to the output example that has 5 columns and 2 rows.

Then, we will check the satisfiability of ϕ using `z3`. If ϕ is *unsatisfiable*, that means the partial program P is *not* consistent with our input-output example, therefore, it's safe to prune P away and not add it into the worklist. If ϕ is satisfiable, we have to conservatively add P in the worklist, since it *may* be correct.

In this assignment, you will implement a `prune` method that generates a constraint for each partial program and checks its satisfiability. You will write your code in `Synthesizer2.java`. It already provides `operatorToSpec` that maps each operator to its specification. You will use this map to implement the `prune` function. Note how this `prune` function is used in the `run` function: it is called only if `attemptToPrune` returns true. The reason we have a check there is because we may not want to perform deduction for *every* partial program we encounter. For instance, consider the following partial AST P'.



P' is *different* from the previous partial program P. However, the constraint for P' is the *same* as that for P, because the way we encode the specifications doesn't really care about whether certain arguments, such as *tmp1*, are concrete or not. In this case, if we were not able to prune P, we *automatically* know that we wouldn't be able to prune out P' either. Note that we can still perform the satisfiability check for P', but that would incur additional overhead since calling `z3` is not always cheap.

In this assignment, you will implement an `attemptToPrune` function which performs pruning *selectively*. In particular, a heuristic we will use is to perform pruning only for programs whose *leftmost path* is complete and all other nodes are `?`'s. For instance, `attemptToPrune` would return true for P since P's leftmost path, *gather* $\rightarrow$ *unite* $\rightarrow x$, is complete (i.e., none of the nodes along that path is `?`) and all the other nodes in P are `?`'s. On the other hand, `attemptToPrune` should return false for P', because of the *tmp1* node, although P' has a complete leftmost path.

4 Test `Synthesizer2`

First, test your `Synthesizer2` using `Test6` and take a screenshot of its output. Make sure `Test2` from Assignment 1 is also in the project, since `Test6` extends `Test2`.

Then, test `Synthsizer2` using `Test7` and observe its behaviour/output. In particular, `Test7` contains a slightly more complex CFG which would make the synthesizer run significantly slower. Consider setting a timeout, such as 1 hour, since it may also take `Synthesizer2` a while to finish.

5 Submission

Your Task. There are five tasks in this assignment.

- (10 points) Task 1: Implement `Test5` and take a screenshot of its output.
- (60 points) Task 2: Complete `prune` and `attemptToPrune` in `Synthesizer2`.
- (5 points) Task 3: Run `Test6` and take a screenshot of its output.
- (5 points) Task 4: Run `Test7` and observe its behaviour/output.
- (20 points) Task 5: Describe and explain what you observed when running `Test6` and `Test7`.

Canvas. Submit the following in one single .zip file:

- **Your code** for Task 1 and Task 2, including **only** `Test5.java` and `Synthsizer2.java`.
- **Your report** in PDF that includes your screenshots and your explanations.

References