HW2 Problem Set

ECS 261

Due: Monday, May 5, 2025

1. Say that a specification is "expressible in Z3" if it can be written as a Python program `def spec(prog):` that takes as input a string variable representing the source code of a program `prog`, and returns a Z3 formula. The specification is true exactly when the Z3 formula is valid. Show that for any two different programs `prog1` and `prog2`, they are distinguishable: there exists a spec which `prog1` satisfies and `prog2` does not. Implement a function that realizes your answer. It should take as input two strings, and return the spec in the above form, i.e. a function from string variables to Z3 formulas. Add a unit test with an example of two particular programs for which the Z3 spec works.

   ```
   def true_of_prog1_but_not_prog2(prog1, prog2):
       # Return a spec in the above form, i.e. return a function
       #     def spec(prog): ...
       # where prog is a Z3 string variable, and which returns a formula
       # valid for prog1 but not for prog2.
       raise NotImplementedError

   def test_true_of_prog1_but_not_prog2():
       raise NotImplementedError
   ```

   As your answer to this problem, explain how your solution works (in a paragraph) and report the results of what happens when you run it, as well as any other interesting observations. Attach your code with your submission.

   Note: This problem shows that if we were to define the partial order of programs such that `prog1` is a "refinement" of `prog2` if all specs which `prog1` satisfy also satisfy `prog2` (similar to stronger and weaker specs, but for programs), this partial ordering would be trivial: it would just be a set of discrete points, no two points comparable.

2. Consider the theory of integers and real numbers (under basic arithmetic) that we saw in class. Recall the syntax as follows:

   ```
   Var ::= n1, n2, n3, ...
   Expr ::= Expr + Expr | Expr - Expr | Expr * Expr
           | Var | 0 | 1
   Formula ::= Formula v Formula | Formula ^ Formula | !Formula
           | Expr == Expr | Expr < Expr
   ```

   (a) Give an example of a formula that is satisfiable over the real numbers but not the integers.

(b) Prove formally that every expression satisfiable over the integers is satisfiable over the real numbers.

(c) Define the theory of truncated 32-bit integers as follows: again we use the same grammar, but now integers are forced to be between $\texttt{MIN} = -2^{31}$ and $\texttt{MAX} = 2^{31}-1$, inclusive. If an expression goes out-of-bounds, we wrap around to the maximum value: for example, $2^{30} + 2^{30} = \texttt{MAX}$ as this is the maximum value available. Similarly $2^{16} * 2^{30} = \texttt{MAX}$, $(-5) - \texttt{MAX} = \texttt{MIN}$ and $(-2^{20}) * (-2^{20}) = \texttt{MAX}$.

Is every formula satisfiable over the integers satisfiable over this theory? Is every formula satisfiable over this theory satisfiable over the integers? Justify both answers.

3. In class, we mentioned that the theory of strings in Z3 – with concatenation, length, and regular expression constraints – can be used to precisely define constraints like "any string matching a US phone number."

Define a function which returns a formula valid exactly for valid dates in `dd-mmm-yyyy` format in the Gregorian calendar. Your function should take as input a Z3 string variable (like `z3.String("s")` – note that this denotes the string variable with variable name `s`, not the literal tring `"s"`) or a string expression (like `z3.String("s1") + z3.String("s2")`). It should output a formula which is true exactly for strings which are valid dates on or after January 1, 1600 in the form `dd-mmm-yyyy` (i.e., two digits for the day, three letters for the month in lower case, and four digits for the year). Test your regular expression first by putting in several constant string regexes (at least 5 positive and 5 negative examples) and second by using it to generate an arbitrary regex in the current year 2025.

Both tests should pass in a reasonable amount of time (under 30 seconds). If they are taking longer, you may need to play with the encoding to get Z3 to work more efficiently. Come to office hours if you get stuck!

```
def valid_gregorian_date(s):
    raise NotImplementedError

def test_valid_date_string_1():
    # Use the function above to test specific valid date(s) and
    # invalid date(s)
    raise NotImplementedError

def test_valid_date_string_2():
    # Use the function to generate a date in the current year 2025
    raise NotImplementedError
```

As your answer to this problem, explain how your solution works (in a paragraph) and report the results of what happens when you run it, as well as any other interesting observations. Attach your code with your submission.

Hints: For help with Z3 regular expression syntax, it will be useful to refer to the the regex help notes[1] in the course repository. You may also want to review the rules for leap days; see Wikipedia.[2]

[1] `https://github.com/DavisPL-Teaching/261/blob/main/lecture2/extras/regex_help.md`
[2] `https://en.wikipedia.org/wiki/February_29`

4. Consider the grammar for the theory of strings (the basic version that just includes string variables, concatenation, character constants, length constraings, and regex constraints). Suppose we were to add an additional construct to this theory:

   ```
   PrefixOf(StrExpr, StrExpr)
   ```

   that evaluates to true iff the first string expression is a prefix of the second one.

   Prove that this would not increase the fundamental expressiveness of our theory by showing that any formula φ with `PrefixOf` can be reduced to a formula φ' without `PrefixOf` that is equivalently satisfiable. That is, φ is satisfiable iff φ' is satisfiable.

5. *Symbolic execution* is a static analysis technique where we "run" a program by evaluating expressions as symbols and formulas, instead of using concrete values. For example, the function in Python `def add(x):` with body `return x + 7` would normally be evaluated by plugging an integer like 10 and returning 17. With symbolic execution, instead we plug in the value `z3.Int("x")` and return the value `z3.Int("x") + 7`, that is we return the symbolic integer expression $(x + 7)$. We can then answer questions aout the behavior of the function, for example, "does there exist an input for which the output is exactly 10?" or "does there exist an input for which the output is out of bounds for an array?" by making an appropriate query to Z3.

   Use Z3 to decide whether the following has an infinite loop bug. The program has 4 paths leading up to the potential infinite loop; you don't need to implement a symbolic execution engine, and you don't need to write an algorithm to compute the answer (for an arbitrary input program); just write down the Z3 formula along each path for this particular program.

```
UNITED_STATES = 1
CANADA = 2
MEXICO = 3
BRAZIL = 4

def add_user(username, location_id):
    # username: a username of type String
    # location_id: an integer
    if username == "":
        print("Error: empty username")
        return
    if location_id < 0 or location_id not in [
            UNITED_STATES, CANADA, MEXICO, BRAZIL
        ]:
        print("Error: country not supported")
        return

    if len(username) > 50:
        user_token = (
            "<user>" + username[:47] + "..." + "</user>"
            + "<location>" + str(location_id) + "</location>"
        )
```

```
    else:
        user_token = (
            "<user>" + username + "</user>"
            + "<location>" + str(location_id) + "</location>"
        )

    while "<script>" in user_token:
        print("Error: invalid user token; please try again")

    DB_HANDLE.get_database(location_id).add_user(username, user_token)

def test_symbolic_execution():
    # Put your symbolic path formulas and assertions here
    raise NotImplementedError
```

Report your answer:

(a) Explain your solution (in a paragraph) and attach your code. Were there any parts of the above code you found especially difficult to encode symbolically?

(b) Did Z3 find an infinite loop bug, prove that no such bug exists, or was it inconclusive (UNKNOWN on one or more paths)?

(c) How long did you find it took Z3 to solve all paths?

(d) Suppose you wanted to include all paths in the same Z3 formula to reduce the number of queries to Z3. (This can be helpful for performance in some cases.) How would you propose doing so with the above technique? Sketch your answer by providing a single Z3 formula that would result from this solution.

Submission instructions:

- Upload your solutions (as a PDF) and your code (in Python) in Gradescope.
- If you use this LaTeX template to create your solutions, please remove the problem statements and include only your solutions.
- Your code should be a file `hw2.py`, together with any necessary helper files. It should include your solutions for parts 1, 3, and 5.
- Please include all of the function names and signatures as listed in the document above, and do not modify any function signatures. (You are welcome to add additional functions and tests.)