Logic and Computability SS23, Assignment 1

Due: 17. 03. 2023

In order to get started with the programming exercises install the `z3-solver` package via pip:

- `pip install z3-solver`

If you have issues with the installation please use the discord channel to ask for help.

We will provide you with the skeletons for the programming exercises so that you only have to do the SMT encoding and testing. Please do not change major parts within those skeletons without consulting your tutor first.

1. Start by cloning the following repository:

   ```
   git clone https://git.pranger.xyz/sp/LAC-Practical-Assignments-2023 --origin upstream
   ```

2. Each weeks exercises will be in their respective directory. Start with the exercises in the `Assignment1` directory.

3. After the deregistration deadline is over, you will receive an email with your personal submission repository. You can add the remote repository with the following command:

   ```
   git remote add origin https://git.teaching.iaik.tugraz.at/lc23/lc23gXX.git
   ```

   Be sure to replace `XX` with your respective ID that you will receive by mail.

4. After you have finished solving the exercises commit your changes and push them to your submission repository:

   ```
   git push origin main
   ```

   You do not need to tag your final commits, we will check out your last commit before the deadline.

5. Additionally to uploading your solutions via git you need to tick which exercises you have solved in the TeachCenter course.

For the first two tasks you will use z3 to prove a simple equality of two propositional formulas and prove correctness of a different approach to compute the minimum of two integers. For the last task on this assignment sheet you will use z3 to solve the *magic square* problem.

1. [3 Points] **Transposition.** We want to prove the following equality:

$$(p \to q) = (\neg q \to \neg p)$$

Your task is to declare necessary z3 variables and add the needed constraints to the solver, such that it checks for equivalence.

The source code skeleton for this exercise can be found in `transposition.py`:

```
# coding: utf-8
from z3 import *

# Create an instance of a z3 solver

solver = None

# Declare needed z3 variables

# Add the needed constraints to check equivalence of the two formulae.

# solver.add(...)


# Check and print the result.

result = solver.check()
print(result)
if result == sat:
    print(solver.model())
```

2. [3 Points] **Branchless Minimum.** Branching can be expensive for the CPU[1]. We want to use a bithack to compute the minimum of two integers by directly applying the result of the comparison, i.e. `slt tmp,rx,ry`.

   We want to prove that the following statements compute the minimum of x and y.

   ```
   tmp = x < y ? 0x1 : 0x0      # slt tmp,rx,ry
   min = y ^ ((x ^ y) & -tmp)
   ```

   Note: This is a unary minus, not a bitwise toggle!

   Your task is to declare necessary z3 variables and add the needed constraints to the solver, such that it checks for equivalence.

   The source code skeleton for this exercise can be found in `branchless_min.py`:

```python
# coding: utf-8
from z3 import *

# Create an instance of a z3 solver

solver = Solver()

min_ite = BitVec("min", 32)
x = BitVec("x", 32)
y = BitVec("y", 32)
# Declare additional z3 variables

solver.add(min_ite == If(x < y, x, y))

# compute the result of slt and store it in a variable
# solver.add(...)

# compute min using the formula
# solver.add(...)

# Check and print the result.
result = solver.check()
print(result)
if result == sat:
    print(solver.model())
```

[1] The Cost of Branching

3. [5 Points] **Magic Square.** For the last exercise we will have a look at the classic pen and paper puzzle called magic square. The premise of the game is very intuitive. You are given a grid of numbers, where some cells are not filled out. Your goal is to determine the missing numbers so that the sum in each *row*, *column*, and *diagonal* is the same. In a sense, you are trying to *balance* out the square of numbers.

   The magic square is read from a file, where the numbers are separated by spaces and unknown cells are marked with ”_”. For the input file 'test0.txt' the resulting magic square is shown in Figure 1.

   Note that the size of the grid is not fixed but may vary, whilst always going to be square.

		28
	35	21
	7	

Figure 1: Example of a magic square.

   Our goal is then to find the missing cells and fill out the square. This is where z3 can help us! An example solution is shown in Figure 2.

14	63	28
49	35	21
42	7	56

Figure 2: Solution of the example magic square.

   Here is a short list of tasks you need to implement in `square.py` when solving this programming exercise.

   (a) Create a Z3 integer variable for each of the cells in the magic square. You should probably give them meaningful names like `C_0_0`, `C_1_2` or similar. This will make it easier to debug.

   (b) Enforce that the known numbers have the expected value. This one is self explanatory, since we cannot change the value of the predefined cells.

   (c) Create a Z3 integer variable which will hold the sum. This variable is used to enforce that all the rows, columns and both diagonals add up to the same number.

(d) Enforce that all columns add up to the sum variable.

(e) Enforce that all rows add up to the sum variable.

(f) Enforce that both diagonals add up to the sum variable.

The template handles the parsing of the input file, and the printing of the solved square.