Logic and Computability WS24, Programming Assignment

Due: 12. 01. 2025, 23:59

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

1. Start by cloning the following upstream repository:

   `git clone TODO --origin upstream`

2. The submission will be handled via your personal submission repository. Add the remote repository with the following command, where you have to replace `XX` with your group number:

   `git remote add origin git@git.teaching.iaik.tugraz.at:lc2425/lcws24gXX.git`

3. 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.

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

You may work in groups of 2 students for this part of the practicals. If you work in pairs, we still ask you to **upload the solutions to both of your personal repositories**. Additionally, please enter your details in the README of both repositories.

1. **[2 points] Burglars**

   We have given the statements of three suspects:

   (a) Ed: "Fred did it, and Ted is innocent."

   (a) Fred: "If Ed is guilty, then so is Ted."

   (a) Ted "I'm innocent, but at least on of the others is guilty."

   Your task is to use z3 and the provided skeleton file `burglars.py` to find out who is guilty and who is not.

2. **[5 points] Branchless Minimum** We can use a bithack to compute the minimum of two integers by directly applying the result of the comparison, i.e. `slt tmp, rx, ry`.

   In order to verify this approach, we need 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`.

3. **[7 points] Magic Square** For this 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 square grid of numbers, where some cells are left blank. Your goal is to determine the missing numbers, such that the sum in each *row, columns* and *diagonal* is the same.

   The magic square is read from a file, where the numbers are separated by spaces and unknown cells are marked with "_". For the `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 the SMT solver 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.

   The template handles the parsing of the input file and the printing of the solved square. The source code skeleton for this exercise can be found in `square.py`. Your task is to add the needed constraints, such that z3 provides a solution via the found satisfying model.

   To run your solver for an input you can simply pass it as an argument: `python3 square.py test0.txt`.

4. **[8 points] Seating Arrangement Problem** For this programming exercise we will take a look at the seating arrangement problem. The problem that you are facing is the following: There are only a few hours left before your weeding and you have lost the seating plan for the big table. You and your friends have gathered a list of all to be seated at the big table, but you cannot simply place them in any order since this could lead to some unpleasent situations. Alongside the list of members at the table you have also come up with a pairs of people which need to be seated next to each other and some which need to have some other guests in between.

   For this exercise we will use our knowledge about custom z3 datatypes and make use of *uninterpreted functions*.

   The skeleton file `seating-arrangement.py` prepares the parsing of the input file and provides a visual representation of your solution. An input file consists of multiple lines which may either have

   - a pair of friends: `Bob likes Alice`,
   - a pair of foes: `Ada dislikes Bob`,
   - a guest without preferences: `John` or
   - a comment: `#John likes Ada`.

   An example input is shown in Figure 4. The corresponding seating plan is shown in Figure 4.

   In order to complete this exercise, you first need to extend the parsing in the skeleton, such that that the prepared data structures get filled with z3 variables. You will then use an uninterpreted function that maps guests to integers, representing the seating at the table. Using this function, we can determine whether two guests would be seated next to each other (`neighbours(a,b)`). Note, that the table wraps around and position `0` is right next to position `len(guests)`. Furthermore, we can add constraints, that limit the solver to place exactly one guest at a given seat. Finally, we can use `neighbours(a,b)` to tell z3 which guests need to be seated together and which must not be seated next to each other.

   If we can find a proper seating plan which adheres to our given constraints we will visualize it, otherwise z3 will tell us that there is no satisfying assignment. Note that z3 will not fully define our uninterpreted function, meaning that we need to extend our call to `model.evaluate(...)` to `model.evaluate(...,model_completion=True)`, which will assing all free variables. The seating plan will be visualized with a table in your terminal and as a list of the mapping in the very end of our program call.

```
Patric dislikes Ada
Patric dislikes Katie
Patric dislikes Bob
Ada
John likes Alice
Bob likes Andrea
Andrea
Alice
Ada likes Julia
Ada likes Katie
Robert
```

Figure 3: The constraints and member list that you have come up with.

```
        Patric Robert
        ---------------------
 Andrea|<<<<<<<<<<<<<<<<<<<<|Katie
       |>>>>>>>>>>>>>>>>>>>>|
    Bob|<<<<<<<<<<<<<<<<<<<<|Ada
       |>>>>>>>>>>>>>>>>>>>>|
   John|<<<<<<<<<<<<<<<<<<<<|
        ---------------------
        Alice Julia

 Seating plan:['Patric', 'Robert', 'Katie', 'Ada', 'Julia',
 'Alice', 'John', 'Bob', 'Andrea']
```

Figure 4: A possible seating plan for the given input file from Figure 1.

5. **[8 points][BONUS] Rock-Paper-Scissors-Spock-Lizard** For this task you need to model the computation of a *linear-congruential generator*. If you want to read more about linear-congruential generators, please consult the wikipedia. Our opponent, the computer, computes its choice by first computing a pseudo-random number s_i that is based on the previous result s_{i-1}:

$$s_i = 11 \cdot s_{i-1} + 12345 \ \& \ \text{0x7fff}.$$

Its choice c_i is then computed with the remainder modulo 5:

$$c_i = s_i \bmod 5,$$

where

- $c_i = 0$ means Rock,
- $c_i = 1$ means Paper,
- $c_i = 2$ means Scissors,
- $c_i = 3$ means Spock, and
- $c_i = 4$ means Lizard.

Your task is to model the computation of each s_i, starting from s_1, and use your knowledge about the computer's choice to tell z3 that $c_i == s_i \text{mod} 5$. Note that you do not know the value of s_0 in this scenario, the problem would become trivial if you would have that information!

The skeleton code can be found in `rpssl.py`. The snippet contains a class `RPSSLComputer` that reads a seed value (`s_0`) and continuously returns the next computer's choice by calling `compute_choice()`.

You execute the script by passing it a value for the seed: `python3 rpssl.py <seed>`.

After you have implemented the correct constraints, your script will win continuously after a few rounds, independent of the seed value.