— Exercise 2 —

Deadline: October 23rd

Task 1 - Bug Fixing

Download the source code of a modified version of SMT-RAT from the following URL:
`https://ths.rwth-aachen.de/wp-content/uploads/sites/4/teaching/praktikum_smt_WS1819/smtrat.tgz`

Unzip the archive and compile SMT-RAT. In the resulting executable file (*smtrat*) the strategy *UFDemo* is used. This strategy contains only two modules:

- *SATModule*: Implements a *SAT solver*.
- *UFDemoModule*: Implements a theory solver for the theory of *uninterpreted functions*.

In the following you should only deal with the *UFDemoModule* that contains several bugs that you should find, explain and fix. Different approaches on how to find bugs are suggested in the following tasks. For every bug you should document the following things:

- original source code
- explanation why this leads to an error
- concrete error
- corrected source code

Example inputs can be found in the folder `samples/`.

Disclaimer

Some of the bugs to be fixed, and therefore also their effects, depend on the used system (for example the processor architecture, the compiler version or the memory layout).

Task 2 - GDB

Try to solve some of the examples with the solver. You will observe that the solver crashes on some of them, for example with the error messages `"Illegal instruction"` or `"Segmentation fault"`.

Use *gdb* to locate and fix the bugs. Therefore you start the solver with *gdb* using the command

```
gdb --args ./smtrat <file>
```

and then start it in *gdb* with the instruction

```
run
```

After the program crashes you can get a *backtrace* with the command

```
backtrace
```

Task 3 - Valgrind

Even though the solver is now able to solve some of the examples there are still other bugs. A common symptom of programming errors are so-called *memory leaks*. These are objects for which memory is reserved in the course of the program execution, but which are never released again. Especially on examples with a long running time this may cause the program to consume more memory than necessary and therefore to crash unnecessarily, as there is no more memory available.

Such an error also occurs in the *UFDemoModule*. Use *valgrind* to locate the source of this memory leak. Therefore you start the program as follows:

```
valgrind --leak-check=full ./smtrat <file>
```

Note that some of the bugs you found in Task 2 could also be found with *valgrind*.

Task 4 - Best Practices

The method `isUFInClass()` is stylistically different from the rest of the code. In this method some *best practices* of modern C++ are disregarded, some parts of the code are unnecessarily complicated or even obsolete.

Make a list of things you would do differently about this code. Explain for each of them why the chosen solution is not suitable.

Implement an alternative version of this method.

Tips & Tricks

- First you do not see any line numbers (for example in *gdb* or *valgrind*). Compile SMT-RAT in the debug mode for additional information. Therefore you use *ccmake* to change the option `DEVELOPER` to `ON`. By that additional checks become active, which may change the error message.
- Read a *backtrace* top-down, from the crash towards the start of the program execution. Skip for the moment methods that are not implemented in your code but for example in the standard library.
- Always read the output from *valgrind* top-down. After the first memory error, some program variables usually have invalid values and thus almost inevitably result in further errors.
- Attention: *valgrind* is actually an emulator. Therefore the execution of a program with *valgrind* is a lot slower than with *gdb*.
- Besides memory errors and *memory leaks*, *valgrind* can also search for other errors. *Valgrind* provides tools for memory checking, (cache) profiling, heap profiling as well as data-race recognition.