

ÖREBRO UNIVERSITY

COMPILERS AND INTERPRETERS

Project: Execution time evaluation

Authors:

Linus Baumgärtner
lbaungaertner@culba.de

Marko Reichhart
marko.reichhart@web.de

October 27, 2022

Introduction and used methods

The goal of this project is to evaluate and compare the execution times of programs with the interpreters constructed in lab 6 and lab 7 and a C++ compiled program. There were compared five different execution methods that are:

- Recursive execution of the syntax tree (lab 6)
- Recursive execution of the syntax tree with optimizations
- Execution in a stack machine (lab 7A)
- Execution in a stack machine with optimizations performed on the syntax tree (lab 7B)
- C++ compiled program

To calculate the times needed to execute the source code a helper class `timer` was created, as seen in listing 1. The timer can be started and stopped. When stopped the duration is printed out. This timer is then started in the programs from the different labs before calling the execution function and stopped after it. In case of the recursive execution the time of the `execute` function and in case of the state machine the time of the `sm.run` function is stopped. By this we get the exact execution time.

Listing 1: C++ class timer

```
1 class timer {
2 public:
3     timer() = default;
4     void start() {
5         begin = std::chrono::high_resolution_clock::now();
6     }
7     void stop() {
8         auto end = std::chrono::high_resolution_clock::now();
9         auto dur = end - begin;
10        auto us = std::chrono::duration_cast<std::chrono::microseconds>(dur).count();
11        std::cout << us << std::endl;
12    }
13
14 private:
15     std::chrono::high_resolution_clock::time_point begin;
16 };
```

All execution methods were evaluated by the same input program that can be seen in listing 2. The program was chosen to provide a significantly big number of operations, while also providing opportunities to optimize for the methods that use some form of simple optimization.

Listing 2: Source program that deals as input to compare the execution times

```
1 n = 10000;
2 i = 1;
3 product = 1;
4 while (i<n) {
5     i = i+1;
6     product = product + i;
7     if (10 / 2 * 5 < 3) {
8         b = i/2;
9         c = b*5;
10    } else {
11        b = i*2;
12    }
13 }
14 print(product);
```

To compare the execution times automatically over multiple executions a simple Python3 program was created. The evaluation method of this program `eval_execution_time.py` can be seen in listing 3. It executes each of the methods described above N number of times and then outputs the average execution time of each method.

Listing 3: Python3 method to evaluate execution time

```

1 def evaluate(N):
2     names = ["Recursive execution", "Recursive execution with optimizations",
3             "Stack machine", "Stack machine with optimization", "C++ program"]
4     times = { names[0] : [], names[1] : [], names[2] : [], names[3] : [], names[4] : [] }
5     for _ in range(N):
6         # Recursive execution
7         os.system("./recursive_execution/calc < input > tmp")
8         times[names[0]].append(int(open("tmp", "r").readlines()[1]))
9
10        # Recursive execution with optimizations
11        os.system("./recursive_execution_with_optimization/calc < input > tmp")
12        times[names[1]].append(int(open("tmp", "r").readlines()[1]))
13
14        # Execute with stack machine
15        os.system("./stack_machine/calc < input > tmp")
16        times[names[2]].append(int(open("tmp", "r").readlines()[1]))
17
18        # Execute with stack machine with optimizations
19        os.system("./stack_machine_with_optimization/calc < input > tmp")
20        times[names[3]].append(int(open("tmp", "r").readlines()[1]))
21
22        # Execute in C++ code
23        os.system("./cpp_program/cpp-program > tmp")
24        times[names[4]].append(int(open("tmp", "r").readlines()[1]))
25
26    print(tabulate(times, showindex="always", headers="keys", numalign="center"))
27    print(f"\nAverage of runtime over {N} runs:\n\n" +
28          tabulate([
29              [names[0], mean(times[names[0]])],
30              [names[1], mean(times[names[1]])],
31              [names[2], mean(times[names[2]])],
32              [names[3], mean(times[names[3]])],
33              [names[4], mean(times[names[4]])]
34          ], headers= ["Method", "time in us"]
35          )
36    )
37    print()
38    os.remove("tmp")

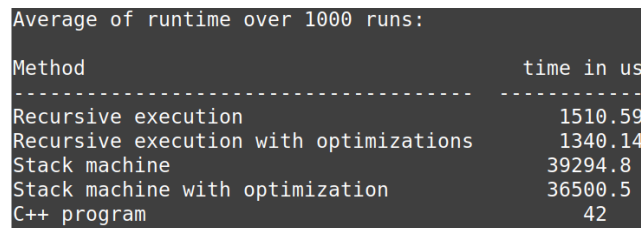
```

Evaluation and Comparison

To get consistent information for the evaluation, each methods was run a number of 1000 times with the command `python3 eval_execution_time.py 1000`. The average of the execution time of the different methods can be seen in figure 1.

As expected the C++ program is executed the fastest with an average of 42 microseconds. The execution time of the recursive execution of the syntax tree is much higher with 1510 microseconds on average. The worst execution time in even a higher magnitude is of the stack machine with 39294 microseconds. This can be attributed to the overhead that is introduced by the stack machine to actually perform a single operation. The recursive execution just uses the basic C++ operations to execute the operations in the leaf of the syntax tree.

Also as expected we get for both the recursive execution and the stack machine a reduction by using the optimizations, in the first case the average time is reduced from 1510us to 1340us by 11.2% in the second case from 39294us to 36500us by 7.1%.



```
Average of runtime over 1000 runs:
Method                                time in us
-----
Recursive execution                    1510.59
Recursive execution with optimizations 1340.14
Stack machine                          39294.8
Stack machine with optimization        36500.5
C++ program                           42
```

| Method | time in us |
|--|------------|
| Recursive execution | 1510.59 |
| Recursive execution with optimizations | 1340.14 |
| Stack machine | 39294.8 |
| Stack machine with optimization | 36500.5 |
| C++ program | 42 |

Figure 1: Average of 1000 runs