LVMD - Final Report

Kristiyan Michaylov
Student Number: 1556274
Email: k.d.michaylov@student.tue.nl

November 9, 2024

1 Introduction

The following report will describe the main decisions taken during the project. It is important to mention that this was the first time I had to design a bytecode interpreter. Therefore, I consulted some literature. The 2 resources which, mostly influenced my design and implementation ideas were the books "Writing an Interpreter in Go" [1] and "Crafting Interpreters" [2]. From them, I received an overall idea of how the implementation should be done and how the process was development process was supposed to happen. Nevertheless, in the end, I would say that my approach was different compared to the 2 books.

2 Architecture

This section will serve as a brief walk-through of my code. In general, my implementation follows the skeleton of the provided VM on GitLab. In the **AstToBciAssembler**, I perform the tree parsing and bytecode generation, therefore the file has almost an identical structure to the initial version. In the **ToyBciLoop**, I execute the generated bytecode instructions, so this file also follows the structure of the initial version. In the **ToyLauncher**, I handle the optimization arguments and also register the functions in a globalScope dictionary variable. Moving to the helper classes I generated, there are several:

- bci.Opcode - where all instruction sets are stored
- bci.Instruction - class for bytecode instructions which contains information regarding where the instruction is stored, what is the opcode of it, what is the variable name, etc.
- bci.Bytecode - this is the list of instructions alongside additional parameters which are generated by the **AstToBciAssembler** and passed to the **ToyBciLoop** for execution. Furthermore, the file contains several storage elements such as *constantPool* where variable/values are registered and *lists for break and continue jumps*. The latter two aid in determining the end location of the jumps.
- bci.GlobalScope - here the whole relevant information about functions (name, number of arguments, rootCallTarget) is added. Information is stored in hashmaps.

3 Bytecode Design

For the bytecode design, I wanted it to be as simple as possible. Since I have no prior experience with this task, I decided to go for the stack-based, instead of register-based, although the latter is faster. Starting with the instruction sets, all of them can be found in the *Opcode.java* class, where they are grouped

by category I don't want to list them here since they will occupy quite some space. Regarding the bytecode, I was influenced by the aforementioned books, however I wanted to have a small number of instructions. It was important to start small and only introduce new opcodes if necessary. Therefore, in the end, by gradual increment, the total number of opcodes is 47. One decision that I think was wrong was to assign to each built-in instruction a specific bytecode command. The bytecode list, as mentioned in the previous section, consists of multiple instructions. Each instruction has 5 properties, namely **opcode, operand, variableName, frameSlot, newVariable**. The first one is for the opcode. The *operand* and *frameSlot* in many cases share the same value. They can point to an index of an element in the constant pool, or a default value (such as 0 or -1) is used in case they are not important. *variableName* is used to indicate what is the name of a variable, or null that is not relevant. Lastly, the *newVariable* indicates if the created variable is new or already defined. This is something which remained in the project but is not used. The main reason for having this format of the bytecode is that I started with it at the beginning. With time passing by, I couldn't find a better format which could help me improve my code.

4 Test outcomes

Concerning the testing outcomes (including benchmarks), the basic implementation and the optimized version both pass 253/271 tests. This result was obtained by enforcing several constraints. For the benchmarks, I have a script which timeouts if 30 seconds have passed without a result. Similarly, for the normal tests, I have a similar thing, however it is 5 seconds. Unfortunately, performance and speed are part of the problems I have with my implementation. I think a big reason for this is that I do not use primitive types more often but rely on classes, such as Bytecode and Instruction. It results in not passing BusyBeaver.sl, Lexicographical.sl, Michel_BinaryToDecimal.sl and 10 of the benchmarks. For the remaining 5 tests, I have errors which are due to not implementing certain functionality, or having a bug. I think if time is not taken into consideration, I won't pass only 5 tests out of the 271 in total.

5 Optimizations

5.1 Implemented optimizations

I solely implemented the **string ropes optimization** which can be found inside the *optimization* package.

5.2 Optimizations implementation

It is used only for string concatenation. Initially, I wanted to implement it also for substrings, however, this would have required significant change in my Toy-

BciLoop, for which, unfortunately, I had no time. The optimization is achieved via the classes:

- Node - top-level specification for the ropes
- ParentNode - class for a node which stores left and right child
- SingleNodeElement - class for storing string data and its length
- StringRopes - class for performing the main work regarding ropes concatenation and conversion to string.

5.3 Performance Impact

The performance impact was measured solely on the SubstringSearch.sl. There, with the optimization enabled, I witnessed a fluctuating improvement between 1 and 15 seconds compared to the vanilla implementation.

6 Reflection

I will use this section to reflect on my experience. Overall, it was a more difficult course than anticipated. Even though I started early, the project had many challenging aspects. Starting from the beginning, how do you design the bytecode? This was hard, and I believe I suffered the consequences of not making the instruction set and bytecode good enough. Another challenge was dealing with functions and recursion. This took more time than anticipated, however in the end I managed to make it work. I think it would have been better to try implementing optimizations alongside the vanilla code. What I did was first to try to pass all the tests and then move to the optimizations. Unfortunately, in the end, I didn't manage to pass all the tests and my implementation was rather slow. I should have made the code more modular since the complexity made adding new functionalities hard. Concerning the course, I think it would be wise to just release the project from day 1 for the next iterations. This way the students will have time to look at it and start thinking early on. I looked at the 2 aforementioned books [1, 2], however without the context, it was difficult to see how things would go.

7 Final Remarks

I had some problems with running the script on the Docker container. I tried to fix it in various ways, however I am not sure if I succeeded. The main issue is related to the line endings of the test files. I deleted all my tests and re-imported them from the Git Lab. The Docker container works fine with my custom shells, you first need to run the test_all and then my shells. I added several scripts, which are used to print all test results (test_all_custom.sh), only failing tests (test_failing), and test-only benchmarks (test_benchmark.sh) all with the aforementioned enforced time constraints.

References

[1] Thorsten Ball. *Writing An Interpreter In Go*. Germany: Self-published, 2016. ISBN: 978-3982016104. URL: https://interpreterbook.com.

[2] Robert Nystrom. *Crafting Interpreters*. United States: Self-published, 2021. ISBN: 978-0990582939. URL: https://craftinginterpreters.com.