Solving the Halting Problem for Brainfuck

Ivan Antonino Arena
DTU Compute
DTU
Copenhagen, Denmark
s233352@student.dtu.dk

Leila Dardouri
DTU Compute
DTU
Copenhagen, Denmark
s231907@student.dtu.dk

Marko Palinec
DTU Compute
DTU
Copenhagen, Denmark
s222979@student.dtu.dk

Daria Yermakova
DTU Compute
DTU
Copenhagen, Denmark
s223047@student.dtu.dk

***Abstract*—In this paper, we investigate the efficacy of employing semantic versus syntactic analysis in addressing the halting problem for programs written in the esoteric programming language Brainfuck**

***Index Terms*—brainfuck, halting problem, semantic analysis, syntactic analysis, abstract interpretation, interpreter, python**

I. Introduction

The ***halting problem*** is a legendary conundrum in computer science, and has long been a symbol of undecidability. It poses a deceptively simple question: can we determine, in a general sense, whether a given program will **halt** or **run forever**? Our project seeks to delve into this enigma by focusing on a particular subset of programs written in the Brainfuck programming language.

Brainfuck is an esoteric programming language created in 1993 by the Swiss computer engineer Urban Müller with the goal of implementing the smallest possible compiler. The language, as a matter of fact, features a very minimal syntax, made up of only eight simple commands. Despite its minimalism, surprisingly enough, Brainfuck still classifies as a **Turing-complete** programming language, being a minor variation of the formal programming language P", which is Turing-complete itself [1].

A. Motivation

The quest to solve the halting problem is not only a fundamental challenge in computer science, but also a symbolic representation of the boundaries of computation. This project offers an exciting opportunity to bridge the gap between theoretical computer science and practical programming.

We will develop a real-world, sound analysis technique that can provide valuable insights into the behaviour of Brainfuck programs, applying the concepts of soundness and completeness in program analysis, while also exploring the limitations of such analyses.

1) Limiting the problem: In Brainfuck, a program operates on an array of memory cells with a small set of commands [2]. In our project we consider a restricted version of the language, which operates on a **finite memory tape**, thereby relinquishing its status as Turing-complete. This constraint allows for a comprehensive analysis of all potential program states and transitions, thus facilitating the examination of the halting problem.

Consider the following Brainfuck program:

Listing 1. Program that outputs 1.

```
1  ++++++++++++++++++++++++++++
2  ++++++++++++++++++++++.
```

It operates by increasing the value stored in the first memory cell 49 times (which is initially equal to 0), then it prints it to screen. The shown output is **1** (the ASCII character corresponding to 49). It is trivial to see that this program **halts**: it takes no input and there are no loops, therefore only a limited amount of instructions to be performed.

Now, consider the following example:

Listing 2. Program that does not halt.

```
1  +[+-]
```

The program increases the first cell by one and enters a loop, which is delimited by the square brackets. The loop's body increments the same cell by one, and decrements it by one. Since a Brainfuck loop only terminates when the value of the memory cell that was being pointed at the beginning of the loop equals to zero, it is immediate to notice that the loop, and hence the program, **will never halt**.

We can abstract what is happening by naming "x" the memory cell on which the program operates, as if it was a variable. We inspect the code one instruction at a time:

1) "+": increment x by 1 ($x = 1$).
2) "[": enter the loop, while pointing at x.
3) "+": the pointer is still on x. Increment x by 1 again ($x = 2$).
4) "-": the pointer is still on x. Decrement x by 1 ($x = 1$).
5) "]": marks the end of the loop. Check if $x == 0$ to exit; $x = 1$ so go back to step 2.

At this point, it is clear that x will never reach the value 0, and the loop will run endlessly, causing the program to never halt. The purpose of our analysis is, indeed, to tell whether a given program ever halts or not.

B. Research question

The main question that our project aims to address is: **Is semantic analysis better than syntactic analysis in solving the halting problem for Brainfuck programs, measured in precision and speed?**

Our work is carried out in the **Python** language, given its simple syntax, the wide availability of tools and libraries, and the fact that every member of the team was already fluent in it.

C. Approach

After collecting a sample set of 25 Brainfuck programs, we begin by tackling the lower-order problems, gradually progressing towards the higher-order ones, as described in the next section. We improve and refine our analysis at each iteration, after thoroughly testing its effectiveness.

Each sample program is processed through a **Python interpreter for Brainfuck**, developed by us. During the initial step, the program's halting behaviour is assessed. If it is determined that the program halts, it proceeds to execution. Conversely, if it is found that the program does not halt, an error message is generated, indicating the non-halting behaviour identified in the evaluation.

1) Syntactic analysis: Consider the listing 1 in section I-A. Again, it is trivial to show that the program halts simply by employing **syntactic analysis** techniques such as **regular expressions** [3] to analyze the code and look for the presence of loops: since the number of memory cells is finite, it is certain that a program without loops is eventually going to **halt**.

Building on top of this, for more intricate problems such as listing 2 in section I-A, we employ **semantic analysis**, specifically **abstract interpretation**.

2) Semantic analysis: In the case of programs featuring non-nested or sequential loops, our initial abstraction involves tracking the concrete value of the cell being pointed to upon entering a loop, disregarding commands unrelated to this variable by abstracting them into a specific symbol. This approach allows us to determine whether the variable eventually reaches the value of 0, leading to the termination of the loop. This analysis already covers an **infinitely large** subset of Brainfuck programs.

When dealing with **nested loops**, it could be necessary to abstract to **multiple variables**, storing their concrete values and abstracting all the other irrelevant instructions with the same symbol. For instance, consider the two-level nested loops provided below:

Listing 3. Program with two-level nested loops and two-variables abstraction.

```
1    +[>>>+[-]]
```

In this scenario, we need to abstract two values as the program points to a different memory cell upon entering the second loop compared to the one accessed during the first loop.
In contrast, in the following example:

Listing 4. Program with two-level nested loops and one-variable abstraction.

```
1    +[>>>+<<<[-]]
```

When entering the second loop, the program points to the same memory cell that was being pointed when we entered the first loop. Therefore, we only need to keep track of the value of this one single memory cell.

We can conclude that our analysis becomes **more complex** as we attempt to solve the halting problem for higher-order subsets of Brainfuck programs.

D. Evaluation strategy

In a preliminary step we **label** the programs collected as ***halting*** or ***non-halting***, running each one of them individually. While our interpreter alone cannot definitively determine if a program will run endlessly, given the relatively small size of the programs in terms of instruction count, we establish a threshold. If a program takes more than **10,000** steps to terminate, we label it as a *non-halting* one. Conversely, programs respecting this threshold are labeled as *halting* ones. After completing the labeling process, we conduct syntactic and semantic analyses, and compare the results obtained with the given label. Simultaneously, we record the **execution time** to measure possible speed differences between the two techniques.

To assess the accuracy of our classification, we employ the **F-score**:

$$F_1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

By comparing the F-scores of syntactic and semantic analyses, we can discern the disparities in accuracy between the two.

It is soon evident that to ensure the development of a valuable analysis of the halting problem, the programs employed need to be reasonably **numerous** and **diverse**, in regards to their complexity.

II. Approach

A. Formal description of the approach and theoretical guarantees

To determine whether semantic analysis is better than syntactic analysis to solve the halting problem for Brainfuck programs, we start with formalizing the Brainfuck language.

We can express the syntax of the language through the following **abstract syntax**:

$$
\begin{array}{lll}
e ::= & > & \text{increment the data pointer by one} \\
| & < & \text{decrement the data pointer by one} \\
| & + & \text{increment the byte at the data pointer by one} \\
| & - & \text{decrement the byte at the data pointer by one} \\
| & . & \text{output the byte at the data pointer} \\
| & , & \text{accept one byte of input} \\
| & [e] & \text{loop}
\end{array}
$$

and, given the tuple $< e, \mathcal{M}, p, ps, d, o >$, where:

- e is the program
- $\mathcal{M}$ is the memory tape
- p is the data pointer, indexing the current cell in the memory tape
- ps is the list of stored data pointers used as guards for loops encountered
- d is the current loop nesting depth in the program
- o is the list of the values outputted by the program

we can define the following **single step semantics** for Brainfuck programs:

- **Increment pointer rule**

$$\frac{p' := p + 1}{< > e, \mathcal{M}, p, ps, d, o > \rightarrow < e, \mathcal{M}, p', ps, d, o >} \; >\text{-rule}$$

When the ">" instruction is encountered, the pointer is incremented by one. Subsequently, the program proceeds to the remaining instructions (represented by e) with the updated pointer p', while leaving the rest of the state unchanged.

- **Decrement pointer rule**

$$\frac{p' := p - 1}{< < e, \mathcal{M}, p, ps, d, o > \rightarrow < e, \mathcal{M}, p', ps, d, o >} \; <\text{-rule}$$

When the "<" instruction is encountered, the pointer is decremented by one. Subsequently, the program proceeds to the remaining instructions (represented by e) with the updated pointer p', while leaving the rest of the state unchanged.

- **Increment memory cell rule**

$$\frac{\mathcal{M}[p] := \mathcal{M}[p] + 1}{< \; +e, \mathcal{M}, p, ps, d, o > \rightarrow < e, \mathcal{M}, p, ps, d, o >} \; +\text{-rule}$$

When the "+" instruction is encountered, the value stored in the memory cell referenced by the pointer, $\mathcal{M}[p]$, is incremented by one. Subsequently, the program proceeds to the remaining instructions (represented by e), while leaving the rest of the state unchanged.

- **Decrement memory cell rule**

$$\frac{\mathcal{M}[p] := \mathcal{M}[p] - 1}{< \; -e, \mathcal{M}, p, ps, d, o > \rightarrow < e, \mathcal{M}, p, ps, d, o >} \; -\text{-rule}$$

When the "−" instruction is encountered, the value stored in the memory cell referenced by the pointer, $\mathcal{M}[p]$, is decremented by one. Subsequently, the program proceeds to the remaining instructions (represented by e), while leaving the rest of the state unchanged.

- **Output rule**

$$\frac{o' := o :: [chr(M[p])]}{< \; .e, \mathcal{M}, p, ps, d, o > \rightarrow < e, \mathcal{M}, p, ps, d, o' >} \; .\text{-rule}$$

When the "." instruction is encountered, the value stored in the memory cell referenced by the pointer, $\mathcal{M}[p]$, is translated into an ASCII character and appended to o, which is the list of previously outputted values. Subsequently, the program proceeds to the remaining instructions (represented by e) with the updated output list o', while leaving the rest of the state unchanged.

- **Input rule**

$$\frac{\mathcal{M}[p] := < input >}{< \; ,e, \mathcal{M}, p, ps, d, o > \rightarrow < e, \mathcal{M}, p, ps, d, o >} \; ,\text{-rule}$$

When the "," instruction is encountered, the value inputted by the user is stored in the memory cell referenced by the pointer, $\mathcal{M}[p]$. Subsequently, the program proceeds to the remaining instructions (represented by e), while leaving the rest of the state unchanged.

- **Loop start rule**

$$\frac{\mathcal{M}[p] \neq 0 \quad ps[d] := p \quad d' := d + 1}{< \; [e]e', \mathcal{M}, p, ps, d, o > \rightarrow < e[e]e', \mathcal{M}, p, ps, d', o >} \; \text{loop-start-rule}$$

When a loop "[e]" is encountered for the first time, we use the value stored in the memory cell currently referenced by the pointer, $\mathcal{M}[p]$, as the guard of the loop. If it is not zero, we store the current pointer, p, in the pointers list ps at position d and increase by one the loop nesting depth. Subsequently, we prepend to the remaining instructions of the program, represented by e', the loop's body, e, and the loop $[e]$ itself. Additionally, we update the state with the adjusted nesting depth, d'.

- **Loop skip rule**

$$\frac{\mathcal{M}[p] = 0}{< \; [e]e', \mathcal{M}, p, ps, d, o > \rightarrow < e', \mathcal{M}, p, ps, d, o >} \; \text{loop-skip-rule}$$

When a loop "[e]" is encountered for the first time, we use the value stored in the memory cell currently referenced by the pointer, $\mathcal{M}[p]$, as the guard of the loop. If it is zero, we skip the loop. Consequently, in the next state, we move to the remaining instructions of the program, represented by e', and leave the rest of the state unchanged.

- **Keep looping rule**

$$\frac{ps[d-1] \neq undef \quad \mathcal{M}[ps[d-1]] \neq 0}{< \; [e]e', \mathcal{M}, p, ps, d, o > \rightarrow < e[e]e', \mathcal{M}, p, ps, d, o >} \; \text{loop-true-rule}$$

When a loop "[e]" is encountered and a value is stored in the list of pointers in correspondence to the preceding nesting depth level (i.e., $ps[d-1]$ is not undefined), it indicates that we are currently iterating within a loop. In this case, the loop guard is determined by $\mathcal{M}[ps[d-1]]$: the value stored in the memory cell referenced by the pointer saved in the list of pointers ps at the preceding nesting depth level, $d-1$, when the loop was entered (i.e., when the "loop-start-rule" was applied). If the guard's value is non-zero, we continue looping by prepending the loop's body, e, and the loop $[e]$ itself to the remaining instructions of the program, represented by e'. The rest of the state remains unchanged.

- **Loop exit rule**

$$\frac{d' := d - 1 \quad ps[d'] \neq undef \quad \mathcal{M}[ps[d']] = 0}{< [e]e', \mathcal{M}, p, ps, d, o > \rightarrow < e', \mathcal{M}, p, ps, d', o >} \text{ loop-false-rule}$$

When a loop "$[e]$" is encountered and there is a value stored in the list of pointers in correspondence to the preceding nesting depth level (i.e., $ps[d']$ is not undefined), it indicates that the loop has already been encountered, and we are currently iterating within it. In this case, the loop guard is determined by the value stored in the memory cell referenced by the pointer saved in the list of pointers ps at the current preceding depth level, d', when the loop was entered (i.e., when the "loop-start-rule" was applied). If the guard's value is zero, we exit the loop. Consequently, we decrease the level of current loop nesting by one and we move to the remaining instructions of the program, represented by e'. Additionally, we update the state with the adjusted nesting depth, d'.

It arises that **loops are the only commands in the Brainfuck language that can potentially result in non-termination.** This observation aligns with the broader theoretical challenges posed by the halting problem.

The **halting problem** is a classic issue in computer science and mathematical logic, initially formulated by Alan Turing (1936). It addresses the fundamental question of whether, given an arbitrary computer program and its input, it is possible to determine whether the program will eventually halt or continue running indefinitely.

The problem is **undecidable**, meaning that there is no algorithm or general method that can determine, for every possible program and input, whether the program will eventually halt or run forever, which poses a theoretical limitation to our analysis.

In our context, we assume a **finite memory tape**. It is crucial to highlight that in this setting, programs without loops will eventually reach a state where no further commands remain to be executed, resulting in termination. Consequently, we classify every **loop-free program as halting**.
Additionally, any program that deviates from the syntax of the language is disregarded. As a result, if a loop is not well-formatted, it triggers the immediate termination of the analysis.

Other theoretical guarantees are established by the previously defined rules. In accordance with the **"loop-start-rule"**, we enter a loop only when the value in the memory cell we are pointing to is not equal to zero. Otherwise, the **"loop-skip-rule"** applies, and the loop will not impact the program's chances of termination. Upon entrance, we store the value of the pointer because the memory cell referenced by it becomes the guard of the loop, and its value must be checked after each iteration. If, after an iteration, the value of the guard has not reached zero, the **"loop-true-rule"** applies and the body of the loop is run again. When the pointed memory cell reaches zero, the **"loop-false-rule"** applies and we exit the loop. Consequently, a **loop runs infinitely many times if the starting pointed cell never reaches zero**.

B. Theory of the used techniques

To analyze the behaviour of the loops in the program, we employ and compare static and syntactic analysis techniques. **Syntactic analysis** involves examining the formal structure of a program without executing it. We aim to identify patterns providing insights into the program's behaviour, specifically related to potential infinite loops or halting.
Since we are not running the program, we cannot compute the value stored in the memory cell pointed when meeting a loop. For this reason, we assume that we enter in every loop present. For instance, the following program:

Listing 5. Program that should halt.

```
1    [+]
```

will be wrongly classified as not halting, even though during an actual execution the loop would be skipped, since the value in each memory cell at initialization time is zero.
Therefore, our focus while performing syntactic analysis is on analyzing the body of each loop, to understand whether it is possible to exit from it once entered.

Regular expressions are a category of syntax analysis techniques, involving sequences of characters defining a pattern to match within a text; in our case, a pattern inside the body of the loop. We initially employ this technique to validate the syntax of the program, ensuring that only characters conforming to the abstract syntax defined earlier are used.

When examining loops, it becomes apparent that a loop with an empty body will not terminate, as it does not affect the value of the guard. Similarly, a loop lacking input or decreasing operations will not halt for the same reason.
This can be easily checked with a regular expression such as `^[^,-]*$`, which asserts that the string does not contain occurrences of "," or "$-$". However, regular expressions alone are **not sufficient for nested loops**. For example, the following program:

Listing 6. Program that should not halt.

```
1    +[>+>+>++>+++[-]]
```

will not be matched by the regular expression in question, and hence will be wrongly labelled as halting. This occurs because a "$-$" is present in the string, even if it is enclosed in a loop on another level, whereas the outer loop is still a non-terminating one.

Another easily recognisable pattern is the presence of the same amount of $+$'s and $-$'s in the loop's body, together with the absence of moving operations. This ensures that, after each iteration, the value of the entering variable will be the same as it was at the beginning.

Semantic analysis involves examining the program's semantics, including variable usage, data flow, and the impact of different statements on the program state. Unlike syntactic analysis, which focuses on the formal structure of the code,

semantic analysis seeks a richer understanding of how the program behaves during execution. Among the semantic analysis strategies, abstracting the program plays a central role.

Abstraction is a formal method used to simplify the representation of program entities and their behaviours while preserving essential characteristics relevant to the analysis goals. The reason behind this simplification is mainly reducing the complexity of a program model, making it more manageable for analysis, without sacrificing the precision required to reason about specific properties.

In our case, we adopt abstraction to make assumptions on the actual value stored by the cell we are pointing to when meeting a loop. This is done by abstracting the guard, and defining a mapping $\alpha : L \to L'$ between the concrete and detailed loop L and the abstract value L' we are defining.

With our abstraction, we ignore every command in the loop that does not affect the loop's guard. Instead of checking for patterns in the body to determine how the instructions will impact the guard, like we did in the syntactic analysis phase, we compute the actual numerical variation in the value of the guard and reason on whether it has a decreasing trend, and will therefore eventually reach zero or not.

Such abstraction aids in refining our analysis while *trying* to ensure **soundness**, meaning that if a loop $\alpha(l) = l'$ halts or does not halt, the concrete version l will also exhibit the same halting or not halting behaviour; **completeness**, meaning that if a loop l halts or does not halt, the abstract version $\alpha(l) = l'$ will also exhibit the same halting or not halting behaviour; and **efficiency**, meaning that it is computationally feasible, practical and understandable.

We analyze loops in the order they are met by the execution, and explore the ones containing nested loops starting with the innermost one and moving outward.

For instance, revisiting the previous example (see Listing 6), we start with the [-] loop, which has as guard a variable of value 3, corresponding to the fourth cell of memory. Upon computing the loop body, we observe a variation of -1. Given this decreasing trend, we assume that it will eventually halt. We then proceed to the outer loop, [>+>+>+++[-]]. The associated variable corresponds to the first cell in the memory tape, initialized with a value of 1. Within the loop, the value experiences no variation (0), leading us to conclude that it will not reach zero. Accordingly, we label the loop as non-terminating and propagate this result to the entire program.

III. Implementation and setup

The source code is available in a GitHub repository at the following address: https://github.com/ivanarena/dtu-02242-project/

All the logic is located in the `src` package, where we find three main classes:

- `SyntaxAnalyzer`, which contains all the logic for the **syntactic analysis**.
- `SemanticAnalyzer`, which contains all the logic for the **semantic analysis**.
- `Interpreter`, which contains all the logic for **interpreting** Brainfuck programs and employing syntactic and semantic analysis.

Additionally, there is a `tester.py` file designed for running the test suite with various output parameters. Both classes dedicated to syntactic and semantic analysis feature an `analyze` method. This method takes Brainfuck code as input, represented as a string, and conducts the respective analysis. If it detects a non-halting program, it raises a `SystemError` exception; otherwise, it returns `True`.

Within the interpreter class, the `interpret` method calls this `analyze` method every time a program is being interpreted. If the analysis does not identify a non-halting program, the interpreter proceeds to interpret the code and prints the output, if any.

The official Brainfuck documentation does not specify how to handle addition and subtraction to memory cells when the value is respectively 255 and 0. In our interpreter, we have implemented absolute lower and upper bounds for every memory cell. Consequently, the interpreter manages these operations by ignoring them: once a cell reaches the value of 0, it is not decremented anymore, and similarly, when it reaches a value of 255, it is not incremented further.

The test suite includes unit tests for both the analysis performed during the interpretation of every program. The results are then evaluated as described in the following section (section IV).

IV. Evaluation and results

In this section, we do an empirical evaluation of our research question:

> *"Is semantic analysis better than syntactic analysis in solving the halting problem for Brainfuck programs, measured in precision and speed?"*

The question can be divided into two sub-questions:

- *Is semantic analysis **more precise** in solving the halting problem for Brainfuck programs?* To which we answer **yes**.
- *Is semantic analysis **faster** in solving the halting problem for Brainfuck programs?* To which we answer **no**.

While not both outcomes favour semantic analysis, the resolution to our primary inquiry inherently resides in the delicate balance between precision and speed. Despite its relatively slower execution compared to its counterpart, our semantic analysis consistently demonstrates notable efficacy in accurately identifying non-halting programs. Consequently, our conclusive affirmation to the main question is **yes**.

A. Experimental Setup

1) Hardware Specifications: The experiments are conducted on a laptop with the following hardware specifications:

- **CPU:** 13th Gen Intel(R) Core(TM) i5-1335U
- **RAM:** 16GB
- **Storage:** SSD 512GB
- **Operating System:** Ubuntu 22.04.3 LTS (Jammy Jellyfish)

2) Software Environment: The experimental setup utilizes the following software components:

- **Programming Language:** Python 3.10 for the experiments.
- **Testing Framework:** PyTest for tests automation and results validation.

3) Brainfuck Interpreter: Our Brainfuck interpreter implemented in Python, described in section III, is employed to perform the analyses and interpret the programs.

4) Benchmark Suite: Considering the specialized nature of our research question and the lack of existing benchmark suites for Brainfuck programs, we specifically designed a tailored collection of benchmark programs. The objective was to cover a spectrum of scenarios, incorporating various syntactic and semantic features. These benchmarks aim to address a diverse range of complexities inherent in Brainfuck programs, both syntactic and semantic.

Further details about the programs are available in the source code and in Table I.

To enhance transparency and facilitate reproducibility, the Brainfuck interpreter code, benchmark programs, and analysis scripts are available in a public repository.

	Program Name	Halting
0	h_01.bf	True
1	h_02.bf	True
2	h_03.bf	True
3	h_04.bf	True
4	h_05.bf	True
5	h_06.bf	True
6	h_07.bf	True
7	h_08.bf	True
8	h_09.bf	True
9	h_10.bf	True
10	h_11.bf	True
11	h_12.bf	True
12	h_13.bf	True
13	nh_01.bf	False
14	nh_02.bf	False
15	nh_03.bf	False
16	nh_04.bf	False
17	nh_05.bf	False
18	nh_06.bf	False
19	nh_07.bf	False
20	nh_08.bf	False
21	nh_09.bf	False
22	nh_10.bf	False
23	nh_11.bf	False
24	nh_12.bf	False

TABLE I
PROGRAMS LIST.

B. Experimental Design

1) Hypothesis Formulation: This research endeavors to examine whether semantic analysis surpasses syntactic analysis in addressing the halting problem for Brainfuck programs, with precision and speed as the primary metrics. The hypotheses are articulated as follows:

- **Null Hypothesis (H_0):** There exists no noteworthy disparity in precision and speed between semantic and syntactic analyses when addressing the halting problem for Brainfuck programs.
- **Alternative Hypothesis (H_1):** A significant distinction in precision and speed is evident between semantic and syntactic analyses when tackling the halting problem for Brainfuck programs.

2) Experiment Execution: The tailored Brainfuck interpreter runs each program through both semantic and syntactic analyses. Detailed data, encompassing execution time and precision, is methodically gathered for each iteration.

Statistical analysis, employing methods like the Student's t-test and McNemar's test, is carried out to evaluate the significance of observed distinctions between semantic and syntactic analyses. The p-value serves as a metric to assess the probability of the observed effects occurring by chance.

C. Analysis of the results

The results indicate that, on average, syntactic analysis is **slightly faster** than semantic analysis in solving the halting problem for the given Brainfuck programs, but the latter **performs much better** in terms of accuracy.

1) Measuring accuracy: To evaluate the accuracy of our analyses, we utilize a widely adopted statistical metric: the **F1-score**. This metric, being the harmonic mean of precision and recall, proves especially fitting for our scenario. It provides a well-balanced assessment by taking into account both false positives and false negatives. Consequently, it enables us to customize our analyses based on the specific cost considerations associated with these data.

In our context, a "**false negative**" occurs when we incorrectly classify a halting program as "non-halting," while a "**false positive**" happens when the reverse takes place. Conversely, our **"true positives"** and **"true negatives"** represent, respectively, programs that are correctly classified as halting or as never halting. These measures are crucial when evaluating the efficacy of a single analysis conducted on a specific sample set of programs.

The results of the syntactic analysis (Table II) reveal that the model attained a precision of 69%, signifying that 69% of the programs predicted as halting were indeed halting. The recall, which measures the ability to correctly identify actual halting programs, reached a rate of 82%. The F1-score reached 75%, indicating the overall effectiveness in predicting halting programs, with an overall accuracy of **76%**.

The macro-averaged metrics take into account both halting and non-halting programs. The weighted averages consider the class imbalance, providing a comprehensive evaluation of the model's performance.

The results of the semantic analysis (Table III) demonstrate a notable precision of 100% for predicting halting programs. This signifies that all programs predicted as halting were verified to be so. The recall achieved a rate of 81%, indicating that 81% of actual halting programs were correctly identified. The F1-score reached 90%, showcasing a high overall effectiveness in predicting halting programs. The accuracy of 88% implies that 88% of all predictions were correct.

Similar to the syntactic analysis, the macro-averaged and weighted average metrics provide a comprehensive evaluation of the model's performance.

	Precision	Recall	F1-Score	Observations
False	0.83	0.71	0.77	14
True	0.69	0.82	0.75	11
Accuracy			0.76	25
Macro Avg	0.76	0.77	0.76	25
Weighted Avg	0.77	0.76	0.76	25

TABLE II
SYNTACTIC ANALYSIS ACCURACY RESULTS.

	Precision	Recall	F1-Score	Observations
False	0.75	1.00	0.86	9
True	1.00	0.81	0.90	16
Accuracy			0.88	25
Macro Avg	0.88	0.91	0.88	25
Weighted Avg	0.91	0.88	0.88	25

TABLE III
SEMANTIC ANALYSIS ACCURACY RESULTS.

Experiment	Syntactic Analysis (ms)	Semantic Analysis (ms)
h_01.bf	0.003	0.004
h_02.bf	0.002	0.002
h_03.bf	0.008	0.005
h_04.bf	0.008	0.005
h_05.bf	0.004	0.003
h_06.bf	0.013	0.008
h_07.bf	0.044	0.015
h_08.bf	0.009	0.007
h_09.bf	0.013	0.007
h_10.bf	0.063	0.169
h_11.bf	0.039	0.015
h_12.bf	0.011	0.007
h_13.bf	0.004	0.004
nh_01.bf	0.006	0.003
nh_02.bf	0.011	0.044
nh_03.bf	0.006	0.005
nh_04.bf	0.007	0.004
nh_05.bf	0.006	0.004
nh_06.bf	0.011	0.007
nh_07.bf	0.016	0.009
nh_08.bf	0.012	0.007
nh_09.bf	0.007	0.004
nh_10.bf	0.008	0.005
nh_11.bf	0.014	0.008
nh_12.bf	0.017	0.007
Average	0.0138	0.0147

TABLE IV
ANALYSES EXECUTION TIMES.

Furthermore, to validate our evaluation, we utilize the **McNemar test**. The obtained **chi-square value of 8.435** with a **p-value of 0.004** indicates compelling evidence of a significant difference in accuracy between syntactic and semantic analyses. Consequently, we can confidently **reject our** H_0 **hypothesis** and affirm the validity of the initial part of our research.

2) Measuring performance: To evaluate the performance of our analyses, we measure the **execution time in milliseconds** of the analysis step, running the test suite on one of our laptops. As all of our devices possess roughly the same computing power, the choice of a specific device is inconsequential, as the resulting measurements are nearly identical. This decision does not introduce bias into the results, given that we are testing both analyses in the same instance. For the purposes of our research, we are considering the results obtained on the machine detailed in subsubsection IV-A1. The results are presented in the following table (Table IV).

The t-test on execution time comparison yielded a **t-statistic of 0.188** and a **p-value of 0.852**. The small t-statistic and high p-value suggest that the difference in execution times between syntactic and semantic analyses is **not substantial nor statistically significant**.

D. Threats to validity

1) Internal Threats: The sole internal threat to the validity of our research pertains to the sample of Brainfuck programs employed for comparison. The scarcity of non-halting programs available on the Internet (as individuals generally refrain from publishing programs that crash or malfunction) and the few constraints imposed on the language and our interpreter, compelled us to create our own set of Brainfuck programs. Despite our efforts to compile a representative collection within the limited time available for the research, we acknowledge that this poses a **limitation** to our work.

2) External Threats: Regarding external threats, our analysis is tailored to our interpreter, developed in accordance with the Brainfuck documentation, which allows for a certain degree of flexibility in implementation (as detailed in section III). Consequently, there exists the possibility of obtaining different results when using a distinct interpreter or a compiler in a **different programming language**. Apart from that, given the experimental nature of our analysis, it is **potentially conceivable** that alternative implementations could yield superior or faster results. However, we maintain confidence in the general superiority of semantic analysis, considering the high syntactic complexity inherent in Brainfuck programs.

V. FURTHER DEVELOPMENTS

A. Advantages and disadvantages of our approach

Naturally, there are alternative program analysis techniques that can be employed to analyze Brainfuck programs. One such example is **dynamic analysis**, which involves observing the behaviour of a program during its execution. While dynamic analysis is potent for capturing runtime characteristics and understanding how a program behaves in real-world scenarios, its application to Brainfuck is hindered by the absence of a standardized runtime environment. This limitation restricts its effectiveness in capturing meaningful runtime data for this particular language. Additionally, a hybrid approach that combines syntactic and semantic analysis might yield superior results, but this lies beyond the scope of our research, as our primary objective is to compare individual techniques.

1) Regular Expressions: Regular expressions exhibit exceptional efficiency in identifying specific syntactic patterns within Brainfuck code, given the language's minimalistic syntax. Their concise and adaptable nature facilitates the straightforward definition of syntactic patterns, aiding in the implementation and maintenance of syntactic analysis. This approach can also be adapted for use with other programming languages. However, regular expressions have limitations when dealing with more complex syntactic relationships or structures in programs. Detecting such structures can be challenging, and it may be impractical to cover all possible patterns within the language.

2) Abstract Interpretation: In contrast to syntax-centric approaches, abstract interpretation aims to understand the semantic meaning of Brainfuck code, an essential aspect in addressing the halting problem. By abstracting the values of the memory cells and closely inspecting each loop, we can conduct a more precise analysis. However, this may become computationally demanding for reasonably large and intricate programs.

B. Future research

The focus on precision and speed in our study has unquestionably yielded valuable insights into the comparative performance of syntactic and semantic analyses for Brainfuck programs. However, acknowledging the multifaceted nature of program analysis, it is imperative to consider a broader spectrum of dimensions for a more comprehensive evaluation. Future research endeavors could strategically explore additional dimensions such as the following:

1) Scalability: scalability is a critical dimension that demands attention. As programs increase in size and complexity, the efficiency and resource utilization of analysis methods become paramount. Investigating how syntactic and semantic analyses scale with the size and intricacy of programs would offer crucial insights for real-world applications.

2) Adaptability to different programming languages: expanding the scope beyond Brainfuck programs to encompass a diverse set of programming languages is crucial. Various languages present unique characteristics, syntax, and semantics. Assessing the adaptability and effectiveness of syntactic and semantic analyses across different languages would contribute to the development of more versatile and widely applicable program analysis techniques.

3) Employment of a different set of metrics: while precision and speed are fundamental metrics, they only scratch the surface of the multifaceted landscape of program analysis. Future research should delve into a broader set of metrics, such as resource consumption or adaptability to dynamic code changes. A holistic evaluation across these dimensions would offer a more nuanced understanding of the strengths and limitations of each analysis approach.

VI. Conclusion

The exploration into whether semantic analysis outperforms syntactic analysis in addressing the halting problem for Brainfuck programs has provided a nuanced understanding of the delicate balance between precision and speed. Our research aimed to illuminate the effectiveness of these two approaches and untangle their individual strengths and weaknesses. In this concluding section, we encapsulate our findings and underscore key contributions.

Our analysis has unveiled insightful comparisons regarding the performance of semantic and syntactic analyses in the context of the halting problem for Brainfuck programs. Both methods have exhibited strengths and weaknesses, contributing to a more comprehensive understanding of their applicability.

In terms of **accuracy**, our results underscore that **semantic analysis demonstrates superiority over syntactic analysis**. Precision-recall metrics consistently revealed higher precision with semantic analysis, particularly in scenarios where syntactic analysis faced challenges. This implies that semantic analysis offers a more dependable classification of halting programs, thereby reducing the occurrence of false positives.

However, this gain in precision comes at a cost to speed. Syntactic analysis, inherently faster, demonstrated **efficiency in execution time**, outperforming semantic analysis across various Brainfuck programs. The trade-off between precision and speed is undoubtedly a crucial consideration, as the choice between the two approaches should align with the specific requirements and constraints of the application. However, the speed difference between the two analyses on our sample programs is, on average, in the order of 10^{-4} milliseconds —not a very significant margin. This allows us to provide a positive answer to our original research question:

> *"Is semantic analysis better than syntactic analysis in solving the halting problem for Brainfuck programs, measured in precision and speed?"*

Our findings bear noteworthy implications for researchers and practitioners in the realm of program analysis. The recognition of the trade-off between precision and speed provides insightful guidance in the decision-making process when choosing an analysis method. **Applications with stringent runtime constraints may lean toward syntactic analysis**, giving priority to speed. On the flip side, **systems that prioritize precision, particularly those in safety-critical contexts, may favor the use of semantic analysis**.

This research contributes to the ongoing discourse surrounding program analysis methodologies by offering a nuanced perspective on their relative effectiveness. The detailed understanding of these techniques adds to the expanding body of knowledge aimed at improving the reliability and efficiency of program analysis tools.

While our study provides valuable insights, it is essential to acknowledge its **limitations**. The dataset, though diverse, may not comprehensively represent all possible Brainfuck programs. Future research could extend the investigation to

a broader range of programs, ensuring a more comprehensive evaluation of analysis methods.

Furthermore, our study focused on precision and speed as key metrics. **Subsequent research could explore additional dimensions, such as scalability, adaptability to different programming languages, or integration with real-world applications.**

In conclusion, our research has delved into the complex landscape of program analysis, pitting syntactic and semantic approaches against each other in the context of the halting problem for Brainfuck programs. The **balance between precision and speed** has surfaced as a central theme, laying the groundwork for informed decision-making in the selection of analysis methodologies. Moving forward, these findings propel us toward the development of more nuanced, efficient, and reliable program analysis tools. This progression not only advances the field but also empowers developers to adeptly navigate the intricate terrain of program verification.

REFERENCES

[1] Böhm, C, "On a family of Turing machines and the related programming language", ICC Bull. 3, 185-194, July 1964.

[2] "Yet another brainfuck reference", https://brainfuck.org/brainfuck.html

[3] "Solving Balanced Parentheses Problem Using Regular Expressions", https://hackernoon.com/solving-balanced-parentheses-problem-using-regular-expressions-q82w3y4u