Assignment 2

Verification of Turing machine properties

Author: Robin Visser

Supervisor: Peter Soulard

Due: 21 May 2023, 23:59

Teacher: R. Reiffenhäuser

Introduction

In the previous assignment we used *finite automata* to dissect and tokenize *Turing machine execution traces* and verify our first *specification* (Turing machine steps) in those traces. In this assignment, we will continue building our framework for runtime verification of Turing machines by verifying two more specifications based on Turing machine properties. The nature of these new properties requires the we 'upgrade' our *specification language* to the next *automata class*, that is, instead of finite automata we will use *pushdown automata (PDAs)* for verification. The target Turing machine model (deterministic one-tape) and corresponding execution traces remain the same as in the previous assignment, refer back to the respective sections if you require an informal explanation.

Background - Verification and assumptions about Turing machines

As stated in the background section of the previous assignment: before our Turing machine starts, the input-string (e.g. 'abbab') is written on the tape from left to right. In addition the Turing machine's head is placed on the leftmost cell (which contains a left endmarker (⊢) by default). See Figure 1 for an illustration.

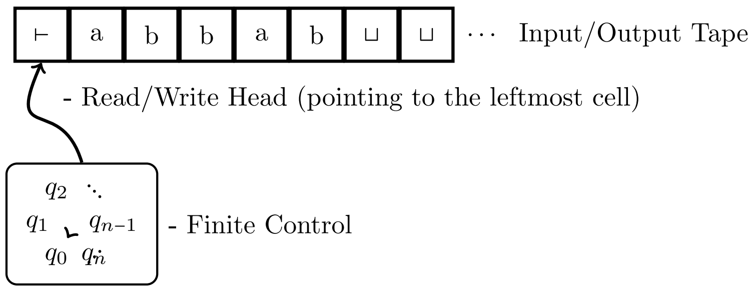
⊢ a b b a b ⊔ ⊔ ···
Input/Output Tape
- Read/Write Head (pointing to the leftmost cell)
q_2 q_1 q_{n-1} q_0 q_n
- Finite Control

Figure 1: Deterministic one-tape Turing machine at the start.

When performing runtime *verification* one must be careful about any implicit assumptions. While Figure 1 gives a nice graphic overview of what the start of Turing machine execution *should* look like, all we have in practice is an execution trace that may or may not represent a valid Turing machine (**and so it may not conform to this 'Proper setup property'**). For now you may *only* assume that any given trace conforms to properties that we have verified previously:

- **Property 1 (Execution trace definition)**: It is a valid execution trace (i.e. consists of events that can be tokenized according to the definition and contains no illegal characters).
- **Property 2 (Turing machine step)**: The tokens (when extracted from the execution trace) form valid Turing machine steps.

Assignment overview

In the file **tokenized_traces.txt** you will find ten tokenized Turing machine execution traces in the style of the previous assignment (with "SPACE" tokens filtered out). The goal of this assignment is to develop pushdown automata that verify correctness of the behaviour described by these traces against a two basic TM properties.

Assignment 2.1 - Verification of Turing machine movement (45 points)

From the previous assignment we know that a valid Turing machine must move its head either left or right (by one cell) at the end of every step. In addition, the *tape* of our Turing machine model (as depicted in Figure 1) is delimited on the left by the *left endmarker* (⊢) and infinite on the right. Given these two statements one might wonder what prevents a Turing machine from moving off of the tape on left side. Indeed, the formal definition by Kozen[1] (indirectly) declares any Turing machine that attempts to step off of the tape invalid. However, nothing is currently preventing this for our execution trace model. For example the tokenized trace "READ LEM WRITE LEM MLEFT" combined with the setup from Figure 1 yields the situation in Figure 2.

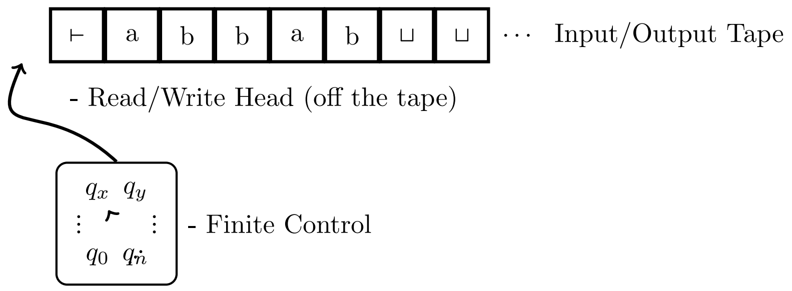

Figure 2: Invalid Turing machine stepping off of the tape

For our framework, we would like to turn the essence of this restriction into verifiable specification without assuming the 'proper setup' from Figure 1 (i.e. no guarantee of either the head pointing to the leftmost cell at the start, or the leftmost cell containing a left endmarker (⊢)). As such we derive the following (depending only on properties 1 and 2):

- **Property 3 (Turing machine movement)**: The machine depicted by steps that make up the (tokenized) execution trace never attempts to move to the left of the cell that it started at.

The goal of the assignment is to create and implement a PDA that verifies whether a given execution trace (that already conforms to properties 1 and 2) complies with this movement property. It is up to you to:

1. Design the PDA and implement it as an instance of the **PDA.py** class in the **verify_movement()** function found in **verification.py**.
    - See the appendix at the bottom of this document for a brief explanation of **PDA.py**.
    - Make sure that it's actually the PDA that is doing the verification (i.e. no manual analysis of the trace). Minor (trace-independent) logic such as always returning the negation of your PDA is allowed.
    - Try to keep the number of states of your PDA to a minimum.
    - Notice that, for this sub-assignment, we are only really interested in the **'MLEFT'** and **'MRIGHT'** tokens at the end of every step.
    - If you don't know where to start, try to tokenize a trace consisting of only a single step (or even a single movement token e.g. ['MLEFT'])
2. Apply the finished **verify_movement()** function to the list of traces (at the bottom of **verification.py**).

The **verify_movement()** function takes a *list of only the tokens* derived from a single trace and returns either True or False, like the following examples:

[1]Kozen, D. C. (1997). *Automata and computability.* Springer.

```
verify_movement(['READ', 'LEM', 'WRITE', 'BLANK', 'MRIGHT', 'READ', 'SYMBOL', 'WRITE',
 ↪  'SYMBOL', 'MLEFT', 'READ', 'SYMBOL', 'WRITE', 'SYMBOL', 'MRIGHT'])
True
verify_movement(['READ', 'LEM', 'WRITE', 'LEM', 'MRIGHT', 'READ', 'SYMBOL', 'WRITE',
 ↪  'SYMBOL', 'MRIGHT', 'READ', 'SYMBOL', 'WRITE', 'SYMBOL', 'MLEFT', 'READ', 'SYMBOL',
 ↪  'WRITE', 'SYMBOL', 'MLEFT', 'READ', 'LEM', 'WRITE', 'BLANK', 'MLEFT'])
False
```

Assignment 2.2 - Verification of Turing machine left endmarker (45 points)

A number of Turing machine properties have to do with the left endmarker. In this sub-assignment we would like to develop *a single PDA* that verifies all of them at once. First, looking at Figure 1, at the start of Turing machine execution the head should point to a cell with a left endmarker written on it.

- **Property 4.1 (Left endmarker initialization)**: The first *symbol* read by a valid Turing machine is a left endmarker (or **'LEM'** token).

Next, the formal definition states that a left endmarker (no matter in which cell) should never be overwritten with another symbol.

- **Property 4.2 (Left endmarker safety)**: Whenever a left endmarker is read within a given step, a valid Turing machine writes back a left endmarker in that same step.

As the name implies, a Turing machine should not move to the left of an existing left endmarker.

- **Property 4.3 (Left endmarker boundry)**: Whenever a left endmarker is read within a given step, a valid Turing machine moves to the right at the end of that step.

Finally, since the left endmarker is part of our Turing machine's *tape alphabet* it may write new left endmarkers in any cell reached by the head (even though Turing machine definitions usually speak of '*the* left endmarker'). However, it is not part of the Turing machine's *input alphabet*, leading to the following:

- **Property 4.4 (Left endmarker consistency)**: Whenever a left endmarker is read from a cell that is *not* the starting cell then it must have been written to the tape by the Turing machine earlier in the trace.

This last property requires that properties 4.1, 4.2 and 4.3 are already (in the process of) being verified by the PDA. It also requires the PDA to keep track of position (you may assume that proper movement has already been verified in the previous assignment).

The goal of the assignment is to create and implement a single PDA that verifies whether a given execution trace (that already conforms to properties 1, 2 and 3) complies with all left endmarker properties. It is up to you to:

1. Design the PDA and implement it as an instance of the **PDA.py** class in the **verify_lem()** function found in **verification.py**.
   - See the appendix at the bottom of this document for a brief explanation of **PDA.py**.
   - Make sure that it's actually the PDA that is doing the verification (i.e. no manual analysis of the trace). Minor (trace-independent) logic such as always returning the negation of your PDA is allowed.
   - Try to keep the number of states of your PDA to a minimum.
   - Notice that the first three left endmarker properties consist of pattern matching that could be verified by a finite automata, only the last property requires a PDA.
   - If you don't know where to start, try to verify a trace consisting of a single valid step (['READ', 'LEM', 'WRITE', 'LEM', 'MRIGHT']).

2. Apply the finished **verify_lem()** function to the list of traces (at the bottom of **verification.py**).

The **verify_lem()** function takes a *list of only the tokens* derived from a single trace and returns either True or False, like the following examples:

```
verify_lem(['READ', 'LEM', 'WRITE', 'BLANK', 'MRIGHT'])
False
verify_lem(['READ', 'LEM', 'WRITE', 'LEM', 'MRIGHT', 'READ', 'SYMBOL', 'WRITE', 'LEM',
 ↪  'MLEFT'])
True
verify_lem(['READ', 'LEM', 'WRITE', 'LEM', 'MRIGHT', 'READ', 'SYMBOL', 'WRITE', 'LEM',
 ↪  'MRIGHT', 'READ', 'SYMBOL', 'WRITE', 'SYMBOL', 'MLEFT', 'READ', 'LEM', 'WRITE',
 ↪  'LEM', 'MLEFT'])
False
verify_lem(['READ', 'LEM', 'WRITE', 'LEM', 'MRIGHT', 'READ', 'SYMBOL', 'WRITE', 'LEM',
 ↪  'MRIGHT', 'READ', 'SYMBOL', 'WRITE', 'SYMBOL', 'MLEFT', 'READ', 'LEM', 'WRITE',
 ↪  'BLANK', 'MRIGHT'])
False
verify_lem(['READ', 'LEM', 'WRITE', 'LEM', 'MRIGHT', 'READ', 'SYMBOL', 'WRITE', 'BLANK',
 ↪  'MRIGHT', 'READ', 'LEM', 'WRITE', 'LEM', 'MRIGHT'])
False
```

Assignment 2.3 - Execution traces as instructions (10 points)

Up until this point we have looked at execution traces as stale reports of Turing machine actions, that is, the properties given test the 'mechanical correctness' of a Turing machine, mostly ignoring input, output and logic (finite control).

A) If your implementation of Assignments 2.1 and 2.2 is correct you should be left with only a single tokenized Turing machine execution trace that adheres to all the properties. You can find the original execution trace in **original_traces.txt** (the line numbers correspond). Assume that the Turing machine which produced this trace operates on bitstrings of length 6 and that the input was "011101". Determine the Turing machine tape output that was produced.

B) What do you think the Turing machine was designed to do? Would it do the same to other length 6 bitstrings? Explain your answers!

Write your answers in **answers.txt**

Submission & Grading

Submit your work to Canvas before **Sunday, 21 May 2023, 23:59** by putting your files in a tarball. *If your code is not written in Python 3 you will receive no points.* In addition, make sure your code:

- Complies with the **PEP8** style guide[2] (e. g. by using flake8)
- Is properly commented

If your program fails to run (or crashes) your work will be graded by judging the code alone, but it will no longer be possible to score all the points.

[2]https://www.python.org/dev/peps/pep-0008/

Appendix - Pushdown automata and PDA.py

The pushdown automaton class in the assignment (**PDA.py**) differs slightly from the one defined in Kozen. Its most important attributes are:

- **PDA.py** is deterministic i.e. **no epsilon transitions**.
- The PDA ignores non-existent relations/transitions (but it will print a warning). So if you are building a PDA that is only interested in two tokens, say **'MLEFT'** and **'MRIGHT'**, then you can (and must) still feed it the entire trace. (Warnings are allowed in your implementation).
- Warnings can be disabled by setting a PDA's 'verbose' argument to False.
- **PDA.py** has no right endmarker, this omittance may trouble 'acceptance by empty stack' implementations. Notice that it is possible to implement (mis)use non-existent/irrelevant transitions to modify the stack.
- The notation for the transitions/relations is the same as in Kozen (minus epsilon transitions); a relation is an element of $(Q \times \Sigma \times \Gamma) \times (Q \times \Gamma^*)$.

  Some examples of correct relations:

```
1: (('S2', 'WRITE', '⊥'), ('S3', ['⊥']))
2: (('Q1', 'MRIGHT', 'MRIGHT'), ('Q2', ['MRIGHT', 'MRIGHT']))
3: (('T', 'BLANK', 'MRIGHT'), ('T', 'ϵ'))
4: (('A', 'BLANK', 'ϵ'), ('B', ['O', 'M', 'G']))
```

Notice that the 'ϵ' in the examples represents either the top symbol of an **empty stack** (example 4) or the empty list of items to push back onto the stack (example 3). The same style of relations is expected by **PDA.py**.

Unicode characters for the initial stack symbol ($\perp$) and epsilon (ϵ) are provided in **verification.py**.