ToC: Final Project [150 points]

Due: Wednesday, December 14th

How to submit (same as for homework)

For all parts of the project that you submit: please put your solution in a folder whose name is your Hawkid. If you worked with a partner please include a file called `partner.txt` that lists the Hawkids for you and your partner. Only one partner should submit. You will both receive the same grade. Once you begin working on the project with a partner, you should continue with them through the end of the project. If you run into difficulties, email Prof. Stump.

How to get help

Please post questions to ICON Discussions, if possible (as opposed to emailing us directly), so that our answers can benefit others who likely have similar questions. You may also come to our Zoom office hours, listed on ICON under Pages - Office Hours.

Overview

You have two options for this project, both related to Turing machines. You are free to pick whichever one you want.

1. **Implementing a TM simulator.** In Section 1, I describe an interface for code which can run nondeterministic Turing machines (so it is a TM simulator). You have to implement this interface in the language of your choice. Please do not use existing implementations of Turing machines that you might find online. You can use standard libraries in the language you choose, but the code for Turing machines should be written by you (and your partner if you have one) from scratch. Then you will demonstrate that your implementation is working through a very small example.
2. **Writing TMs.** In Section 2, I describe a problem that you should solve by writing a Turing machine with my Haskell code (provided along with these instructions). You have to show that your implementation is working on some testcases. (If you cannot get it working perfectly, you can still earn partial credit.)

Rationale

Writing Turing machines is an exercise in very annoying programming, which is challenging and hence hopefully will provide a good workout. So ideally I would just assign option (2). But the only implementation I have right now is in Haskell, and I decided not to require any more Haskell following the mid-semester evaluation. That is why I am including option (1). In future semesters, if I get good solutions for this option, I will be able to have students use those instead of my Haskell

code, for writing Turing machines. And for this semester, it is also an interesting programming exercise to write a simulator for Turing machines. So this is a good option in its own right.

1 Implementing a TM simulator (any language)

Here is a simple interface for simulating nondeterministic Turing machines. The file `MonoTM.hs` that I am including with these instructions provides an implementation of this interface in Haskell, as an example. You can consult that code (or ask a question) if you are unsure of any details. I have highlighted the required functions in `MonoTM.hs` with a widely formatted comment, to help you find them if you wish.

To simplify the code, we will use numbers for the states of the TM, and characters for the letters of the input alphabet and tape alphabet. Your code will need to define some types:

- **TM** for TMs.
- **Config** for configurations, which tell the current state of the TM, the contents of the tape, and the location of the readhead.
- **History**, for a list of lists of `Config`s. You will want to represent a set of configurations that the nondeterministic TM list of `Config`s could possibly reach as a list of `Config`s. A `History` then is meant to represent a sequence of such sets of configurations, that your simulator produces in order.

Using those types, you should then implement these functions:

- **createTM**, which should create a TM from this information:
  - **states**, a list of all the states used in the TM
  - **inputs**, a list of all the input characters used
  - **tapesyms**, a list of all the tape characters used
  - **blank**, the blank character
  - **leftend**, the left end marker (character)
  - **trans**, a list of the transitions of the TM. You will probably want to invent a type `Trans` or something like that, for transitions. Each transition tells the source and target states for the transition (i.e., the states that the transition connects), the symbol that should appear on the tape for the transition to be triggered, the symbol that should then be written on the tape, and finally the direction to move the readhead after that transition is triggered.
  - **start**, the start state
  - **final**, a list of the final states
- **showTM**, which should convert a TM to a string, showing all the data of the TM in some reasonably nice format.

- **showConfig**, which should similarly convert a `Config` to a string
- **showHistory**, which does the same for `History`s. Please separate the elements of a `History` by newlines when you print them, for readability.
- **initialConfig**, which should take a TM and a string of input characters, and return the initial configuration of the machine, where the left endmarker is at very first cell of the tape, then the input string comes next, and the readhead points to the start of the input string.
- **configs**, which takes a TM, a number of steps to run the simulation, and an input string, and returns the `History` representing running the given nondeterministic TM that many steps.
- **accepting**, which takes a TM and an input string, tries to return the first `Config` it finds whose state is a final state.
- **accepts**, which takes a TM and an input string, and returns true if the TM accepts the string, and false if it rejects (and may diverge, of course, if the TM diverges).

You do not have to implement other functions from `MonoTM.hs` like `writeGraphViz` (of course you may do so if you wish, but only the above functions are required).

1.1 Reporting and testing

To get full credit, you must describe your code, and test it. You must implement at least one TM (can be very simple, like tripletm from `MonoTMExamples.hs`, or even simpler) using the interface you implemented.

For documentation, your submission must include a `README.txt` file in plain text (not Microsoft Office or other format) which has these sections:

- **Tooling**: state which programming language you used to implement your solution, and any libraries that need to be installed to run it
- **Sources**: list the source files for your solution
- **Tests**: list files containing tests you used; at least one must be a TM, but you can also have tests for your `initialConfig` function, printing functions, etc.
- **Implementation**: describe briefly how you implemented the simulator (like what data structures you used, etc.)
- **Status**: explain whether or not your simulator is fully functional, has known bugs, etc.

Also, your submission must include a file called `TESTING.txt` that shows the results of your tests. We may not be able to try running your code ourselves, due to time limitations (especially with possibly needing to install new software). So we want you to show us what your code can do, by showing the results of calling the various functions, using the tests you wrote.

2 Writing TMs (Haskell)

Using the provided `MonoTM.hs`, implement a TM for the *addition problem*: given a string x over the alphabet $\{0, 1, +, =\}$, check whether x is of the form $a + b = c$ where a, b, and c are numbers in binary, with least significant digit on the left of the string, such that a plus b indeed equals c using binary addition. You can add your TM to the bottom of the provided file `MonoTMExamples.hs`.

For full credit, your solution should include a file called `README.txt` with these sections:

1. **Idea**: the idea of your TM. Explain the intuition for how your TM works. Also, please pick a small number of states (3 to 5) and explain what is happening at those states. This is just to help us understand your TM better.
2. **Status**: is your TM working correctly as far as you know? Are there any issues?
3. **Testing**: testing you have done. You can show the `History` you get running `configs` some number of steps on your TM, for some example input strings. You should include at least one accepted string and one rejected string. Please show tests you are running together with the output those tests produce. Again, this will help us in case we do not have time to run your code ourselves to test it.