Turing Machine Simulation Language

Isaac McGarvey (iam2108)
Joshua Gordon (jbg2109)
Keerti Joshi (kj2217)
Snehit Prabhu (sap2131)

The Concept

We propose to create a language for simulating Turing Machines. The language will allow the user to specify the elements of the machine: its states, alphabet, and transitions. That machine can then be executed on an input string. The language interpreter will simulate the machine and also provide some way of tracing or visualizing the operation of the machine as it is executing.

Turing Machines

A Turing Machine (TM) consists of a tape and a control mechanism. The tape has an infinite number of cells, each containing a symbol. The set of symbols is finite. The control portion of the machine can be in any of a finite number of states. It always starts in a particular state. It has a transition function that defines how it moves between states. The transition function also determines where to read and write on the tape (i.e. the position of the head) and what to write on it. Some of the states are also accepting (i.e. final) states.

Before the TM begins executing the tape contains an input string. All the other cells of the tape contain a special symbol called the blank symbol. The read/write head of the tape starts at the left non-blank cell. At each stage of execution, the control unit looks at its current state and the symbol on the current cell of the tape. Based on that, it writes a new symbol to that cell, moves one unit left or right, and changes to another state. The transition function is what specifies this mapping. The machine will continue to execute either when it reaches an accepting state or it finds itself in a state for which the transition function is undefined. The output of the machine is the string of non-blank symbols remaining on the tape.

An Example

Shown below is an example of what a program to instantiate a TM would look like. It begins by defining the states and symbols. Then, the bulk of the program is devoted to defining the transition function.

```
Class TM Multiply {

States: q0, q1, q2, q3, q4, q5, q6, q7, q8, q9, q10, q11, q12
Alphabet: 0, 1, X
Start: q0
Final: q12

Transitions :
{
   transition(q0, 0) = (q6, blank, right)
   transition(q6, 0) = (q6, 0, right)
   transition(q6, 1) = (q1, 1, right)
   transition(q7, 1) = (q8, 1, left)
   transition(q8, 0) = (q9, 0, left)
   transition(q8, blank) = (q10, blank, right)
   transition(q9, 0) = (q9, 0, left)
   transition(q9, blank) = (q0, blank, right)
   transition(q10, 1) = (q11, blank, right)
   transition(q11, 0) = (q11, blank, right)
   transition(q11, 1) = (q12, blank, right)

   /* states q1 – q5 act as the “copy” subroutine */

   transition(q1, 0) = (q2, X, right)
   transition(q1, 1) = (q4, 1, left)
   transition(q2, 0) = (q2, 0, right)
   transition(q2, 1) = (q2, 1, right)
   transition(q2, blank) = (q3, 0, left)
   transition(q3, 0) = (q3, 0, left)
   transition(q3, 1) = (q3, 1, left)
   transition(q3, X) = (q1, X, right)
   transition(q4, X) = (q4, 0, left)
   transition(q4, 1) = (q5, 1, right)
   transition(q5, 0) = (q7, 0, right)

} /*end transitions

Func: Run (Input_String)
Event: Visualize ()
Delay: 400

}  /*end class
```

This TM outlined in the example has been taken from a textbook (*Introduction to Automata Theory, Languages, and Computation*, 2nd ed., pg. 335). It multiplies two positive integers. It takes an input in the form 0^m10^n (i.e a sequence of *m* 0's, followed by a 1, followed by a sequence of *n* 0's) and produces an output in the form 0^{mn} (i.e. a sequence of 0's of length *m* times *n*). Notice that the number are represented by the 0's using a unary encoding and the 1 simply serves as a separator. For example, given the input "001000" (2 times 3), the machine will produce the output "000000" (6). The machine accomplishes the multiplication by copying the first operand *n* times. This copying is implemented in states q1 – q5, and is effectively an independent subroutine within the larger machine.

The mandatory function `Run(Input_string)` is a part of the definition of a TM. It accepts an ***input string*** which is checked to verify if it is defined only over the acceptable `Alphabet`. The function always returns an ***output string***, also over `Alphabet`. For simplicity sake, the TMs will use the same `Alphabet` for both input and output.

The keyword `Event` is used to define a type of function that gets invoked after every transition. `Visualize()` in this case is a function that renders the an (ASCII ?) graphical TM on the screen and lets the user step through the transitions one by one. The keyword `Delay` specifies the number of milliseconds to stall the processing after each event call. If not specified, the default value will be used.

The keyword `Class` accepts one of two qualifiers : `TM` or `UTM`, for Turing Machine or Universal Turing Machine respectively.
In the example above, we defined a new Turing Machine T. We could alternatively have also defined :

```
Class UTM Univeral {
    TM : Multiply, Add
    Func : Multiply ( Add( Input_string) )
    Event : Visualize ()
}
```

The UTM definition accepts 2 TMs "Multiply" and "Add", defined elsewhere. In this case the function defined is an aggregate that feeds the output of one TM to the input of another. Ensuring that the 2 machines share common representational semantics is the users responsibility (For ex., if Add represents Integer 1 as 0000 while Multiply represents it as 0001, that is the programmers folly).

The UTM has a distinct visualization. This is because it represents the entirety of the definition of each of its constituent TMs (States, Alphabet, Start, Final, Transitions) as a single large input string.

Curing the "Configuration File" Syndrome

Although the example program above does an actual calculation, it does look a little like a configuration file in that the bulk of the program is spent simply defining the transition table. How can we make this less like a configuration file and more like an actual program? The UTM aspect of the language could solve that problem by allowing TMs to be combined to perform a larger calculation.

One thing we noticed after coming up with this idea is that it is nearly identical to the petri net specification language that is given as an example on the class website in that both languages simply define states and transitions between them. One difference is that the petri net language allows the user to define blocks of Java code associated with a transition that are executed when that transition is taken. We could add a feature like that to our language.

Even if the Turing Machine is mostly just a table, there are a lot of interesting things that could be done with the transition events and combining multiple TMs to add some complexity. There is also the possibility of adding some syntactic sugar to the transition table definition to reduce redundancy. For example, perhaps the user could say something like:

```
transition(state1, *) = (state2, *, right)
```

This could indicate that the state always copies preserves the existing symbol on the current cell. It could even be a little bit like a simple pattern matching system.

Conclusion

One of the advantages of this proposal is that the basic idea is very simple and well-defined. The most basic incarnation of this idea – simply defining the machine and simulating it – would be easy to implement. This would prevent our group from getting in over our heads. However, this idea also allows for adding plenty of more advanced features through the implementation of the transition events and the visualization of the machine's execution.