Turing Language

Developing a Domain-Specific Language in JetBrains MPS

By

Adrian Langemyr, Bjørn Vetle Ledaal and Olav Markus Sjursø

in

IKT445
Generative Programming

Supervised by Prof. Andreas Prinz
Faculty of Engineering and Science
University of Agder

Grimstad, Fall 2019

Acronyms

MPS JetBrain's Meta-Programming System

DSL Domain-Specific Language

GPL General-Purpose Language

JBC Java Bytecode

TM Turing Machine

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

We are overdue for the next big jump in the abstraction of programming languages [1]. Languages started simply being lines of code executed sequentially before it evolved to include functions that were one of the first levels of abstractions. Next classes and objects were created to add a new level of abstraction. This increase in abstraction has historically come with an increase in efficiency and productivity.

The next level of abstraction will be Domain-Specific Languages (DSLs) which are specialized to a particular application domain, contrary to a General-Purpose Language (GPL) which targets a broad range of domains. DSLs can be divided into different groups, which include domain-specific markup languages, domain-specific modeling languages, and domain-specific programming languages.

1.1 Background

This project builds on the results from the two previous groups developing the Turing Language. The first group [2] created a DSL in JetBrain's Meta-Programming System (MPS) able to describe and run a Turing Machine (TM) inspired language, including all aspects of the language except for execution. The developed syntax resemblance the physical components of a Turing Machine. The second group working on the Turing language[3] created a combination machine which can be run trough an interpreter. In their reflection they pointed out the lack of a proper graphical user interface to run the combination machines.

For this assignment, we chose to work on implementing a DSL with the MPS, specifically a language for describing Turing machines. A Turing machine is a model of computation which defines an abstract machine that manipulates a tape of information according to a set of rules. The Turing Machine was invented by Alan Turing in 1936. The machine consists of a tape of infinite length and a finite set of symbols that can be read from or written to the tape.

The two previous groups have implemented a working Combination machine, and a structure for the Table machine. We would like to further improve the result of these groups by making the syntax easier to understand and implement the missing pieces of the Table machine.

1.2 Previous work

Turing Group 2017

Ground work was done to make the TM and text editor work in a predictable manner. For this project only the Table machine was implemented. The Table machine editor was set up to allow for all the required operations, such as read, write, move and next state. The reliance on constrains for the different concepts was a bit misguided, as it does not properly restrict the fields as expected.

Turing Group 2018

While it was possible to create combination machines in the Turing language, table machines were not yet fully implemented. In regards to the table machine's editor; it was not possible to represent the read values from the tape for a given state. It was also only possible to enter the default values for *write* and *move*. There was no visual distinction between keywords, like *State* and constant values like the values written to the tape for a given state. The table machine was also lacking a way to be run, and was missing an *interpret* function. There was also some unused code in the form of Java classes for different concepts.

2 Language

As previously stated, there was no way to represent a read value from the tape to a state in a table machine, so this feature was added to the editor. Keywords were also highlighted with appropriate colors, distinguishing them from variables, and values entered by the user. Just like in the combination machine, a *run* intention was added to the table machine, which triggered an interpreter function. This was mostly based on the corresponding functions found in the combination machine. It is responsible for representing the user input as a Java object (*list<int>*).

The *run* function was based on the behavior of the *ifStatement* and is the function that is called when the user starts a table machine through the intentions menu. It describes what actions will be performed by the Turing Machine (TM) based on what it reads from the tape, and what the current state is. When the machine processes the tape, its content should be altered according to the states defined by the user in the Turing language. The results are displayed in a new window.

In order to create a program in the Turing language, one first defines a machine name. Then the states are defined. Do note that the initial state is not recognized by the program, and therefore it does not run through the consecutive states, correctly processing the tape. A state is defined by giving it a name, and the properties *read*, *write*, and *move*. In addition to these conditional behaviors, a *next* is also defined for each state, referring to the next state. Each state can have multiple *next* states, and this is specified by adding additional state behavior segments. A state can behave in one way if the symbol read from the tape is *0*, and in another if it reads a *1*.

3 Solution

The solution consists of the language structure, syntax and semantics ...

3.1 Structure

Concepts

- State
  A Turing machine has an initial state, subsequent states, and a final state. Each state has a particular response to an input (behavior) depending on the current state of the machine.
- Move
  Upon reading data from the tape, each state has defined a moving procedure for each possible value which is read.
- Read
  Data read from the tape dictates what actions will be performed by the Turing machine.
- Print
  Data can be printed to the tape, based on the current state and action performed.
- Tape
  Tape is a reference to a virtual or physical tape that is seekable back and forth during the Turing machine's operation. Values are read from, and written to the tape as the machine traverses through the states until it reaches a final state.

Constraints

When programming a Turing Machine, there are certain actions that should not be permitted. Applying such constraints can prevent runtime errors, and make sure the program runs as expected.

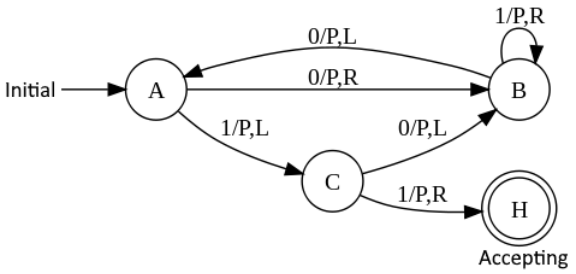

Figure 1: Example state machine

The Turing Machine can have only one initial state. There needs to be a single state in which the Turing machine starts (initial state) for any input. For this reason, another machine should be

introduced to ensure that the first machine starts at either the leftmost, or rightmost value of the tape given any input.

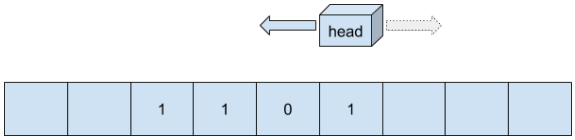

Figure 2: Tape and read head

The Turing Machine must have at least one accepting state, the output is only valid if the Turing Machine stops at an accepting state. Depending on the operation the machine performs (subtraction, sort, etc.) the conditions for reaching an accepting state may differ, but the head of the machine should not be allowed to move beyond the outer values on the tape. Therefore, a "MOVE LEFT" should not be permitted when the head has reached the leftmost part of the input. Likewise, a "MOVE RIGHT" should be impossible when the head has reached the rightmost value. Behaviour in such cases is undefined at this time.

As the machine itself is fairly simple, there are not many constraints to consider. Much of the work lies in the actual implementation of Turing machines that performs various tasks, like mathematical operations, sorting etc.

3.2 Syntax

The existing editor from the 2018 Turing project [3] is hard to understand without being intimately familiar with the project. We would like to improve this syntax by making it more similar to the graphical representation. The syntax should be understandable for any domain expert. The old syntax is represented in Figure 3.

Each state of the machine has a certain defined behavior, which determined its reaction to various input values (1 or 0). The behavior is both a write-, and a move command. If the state is final, then the machine will finish, and not process the tape anymore.

```
Combination Machine SORT

  0 L L right 0 R R SORT_loop
  SORT_loop :
    1 left if 1 : SORT_b1
              0 : SORT_b2
  SORT_b1 :
    0 L L 1 right if 1 : SORT_b3
                      0 : SORT_b4
  SORT_b2 :
    L 1 R R left
  SORT_b3 :
    0 R R SORT_loop
  SORT_b4 :
    1 R R left
```

Figure 3: Existing editor syntax

3.3 Semantics

Transformation

Since the Turing machine is turned into MPS base language in a model-to-model way and then executed rather than transformed into text which is parsed, there is not much to write about transformations.

Execution

Tape; input contents
An object representing the input of the Turing Machine per definition, the length of the tape is infinite, and the Turing machine must be able to recognize the boundaries of the actual input values.

Head; current position
At runtime, head is an object which reads the tape content at its current position and writes a value and moves according to its state.

State; action and head movement for each expected token
At runtime the state will hold information about what tokens correspond to which actions. Actions may be changing to another state or halting. The head movement accompanies the action and may move the head left, right or not at all.

Global; current state(node), halted/running
The runtime needs to keep track of which node is currently being processed, and if the Turing Machine is running or halted.

At the moment, programs written in the Turing machine language can only be run within MPS. However, in order to run it outside intermediate code must be generated before execution. In the case of the Turing language, the code would be first transformed to Java, then it might be further compiled to Java Bytecode (JBC).

3.4 Self Evaluation

Old and dead code have been cleared out to reduce confusion that could arise from inexperienced MPS users. Some outright incorrect code have been undone to make it easier to get an overview of the project. Previously it was unclear which parts of the project was executed, because leftover parts were not properly removed. We have hopefully sufficiently documented the intended input and output of the language, unary, as this caused some initial confusion for our group.

There is still some work left to do in the project before it can be considered complete. Our main goals, the completion of the Table machine and making the syntax easier remains to be completed.

The current project has a decent base, but some MPS experience is a definite plus if further work should be carried out.

Intentions could be organized better. Specifically order of importance, restricting the scope, and expanding some options. For example, a *move* intention should be expanded to a 0 and 1 move intention.

4 Plan and Reflection

4.1 Plan and time spent according to the plan.

Throughout the project timeline the group met on a regular basis, usually once or more per week. All group meetings spanned at least 3 hours, and at most 9 hours. During these meetings all group members contributed towards the completion of the project.

Meetings with the supervisor for this project was done on an as-needed basis, where most meetings took place at the tail end of the project.

4.2 Reflection around the project management and planning.

Throughout the semester we were given assignments to deliver reflections on the different parts of the project.

As an addendum to the above mentioned appendices, we believe our attempt to describe the syntax of the Combination machine was misguided. The current syntax is sufficient and compact enough to be left alone. Any attempt to overhaul this syntax must be done in full, but the compactness is hard to abandon and should be taken into account.

5 Discussion

5.1 Language coverage

In terms of language coverage, the current implementation is set up to be Turing complete. This means that all aspects of a Turing machine are implemented, and can be expressed within the language.

5.2 Choices

It was decided that the alphabet should be limited to 1 and 0, with 1 representing a unary value, and 0 representing a separator/boundary symbol. Although expanding the alphabet to include other symbols would enable the creation of machines that could operate with binary values, it would mostly further complicate the project at its current state. Working on the table machine was prioritized over editing the syntax of the combination machine, as described in "Syntax".

5.3 Future Work

While working on the project, several challenges arose along the way. These should be addressed in later Turing machine projects:

- Table layout in Table Machine editor
- Set initial state for table machine
- Improve the user interface so the program continues running after processing a tape
- Implement a *step* function that would execute one move along with the appropriate related actions
- Replace *read*, *write* and *move* with a single *step* function

References

[1] A. Prinz, *Ikt445 generative programming introduction*, University of Agder, 2019.

[2] E. S. Samdal, R. Isakser, and A. Iyagizeneza, *Turing machines*, 2018.

[3] A. Gammelsrød, J. K. Giang, and B.-I. S. Thoresen, *Interpreter for turing language in jetbrains mps*, 2018.