UNIVERSITETET I AGDER

Interpreter for Turing Language in JetBrains MPS

IKT445 Generative Programming

Authors

Anders Gammelsrød

James Khoi Giang

Bjørn-Inge Støtvig Thoresen

Supervisor

Andreas Prinz

December 5th 2018

University of Agder

Abstract

In this project a language for Combination Turing machines was created in JetBrains MetaProgramming System (MPS). The language allows for a textual notation of Combination Turing machines, and lets the user select any combination machine to run through an interpreter.

Contents

1 Introduction

The task was to create a language for describing Turing machines in JetBrains MPS. This task was previously started by another group before us, but not completed. We were given their report and code repository to use as a reference and starting point. Throughout the project we received a lot of guidance and help from our supervisor Andreas Prinz, whom we would like to thank.

The latest working version of our implementation can be found on the master branch of our Bitbucket repository.

2 Language

Turing machines were first described by Alan Turing in 1936 and comprises of simple abstract computational devices intended to help investigate the extent and limitations of what can be computed. Turing called them "a-machines" (automatic machines) and designed them to compute real numbers. They were first called "Turing machines" by Alonzo Church in a review of Turing's paper (Church 1937)[2].

Today, they are considered to be one of the fundamental models of computability and theoretical computer science. Most computer scientists agree that Turing's formal notation captures all computable problems, despite its simplicity. This means that for any computable problem, there is a Turing machine which computes it[2].

A simple Turing machine consists of a head and a tape of infinite length which acts like the memory in a computer or any other form of data storage. The head is positioned on the tape which is divided into cells which can be blank or contain symbols. A typical 3-symbol Turing machine can process only blank squares and squares containing the symbols "0" and "1". With the head of the machine positioned over a single cell at a time, the machine can perform three basic operations[3]:

- Read the symbol on the cell under the head
- Edit the symbol on the cell by writing a new symbol or erasing it
- Move the head to a cell to the left or to the right

3 Solution

In this section the solution for the language implementation will be discussed. The following aspects of the language will be explained: The structure, the syntax and the semantics. Finally, the pros and cons of the implementation will be discussed.

3.1 Structure

The structure of the Turing language describes how the different concepts are connected. Figure 1 below shows a UML class diagram of the structure. Below the figure is a list of the various concepts and their attributes.

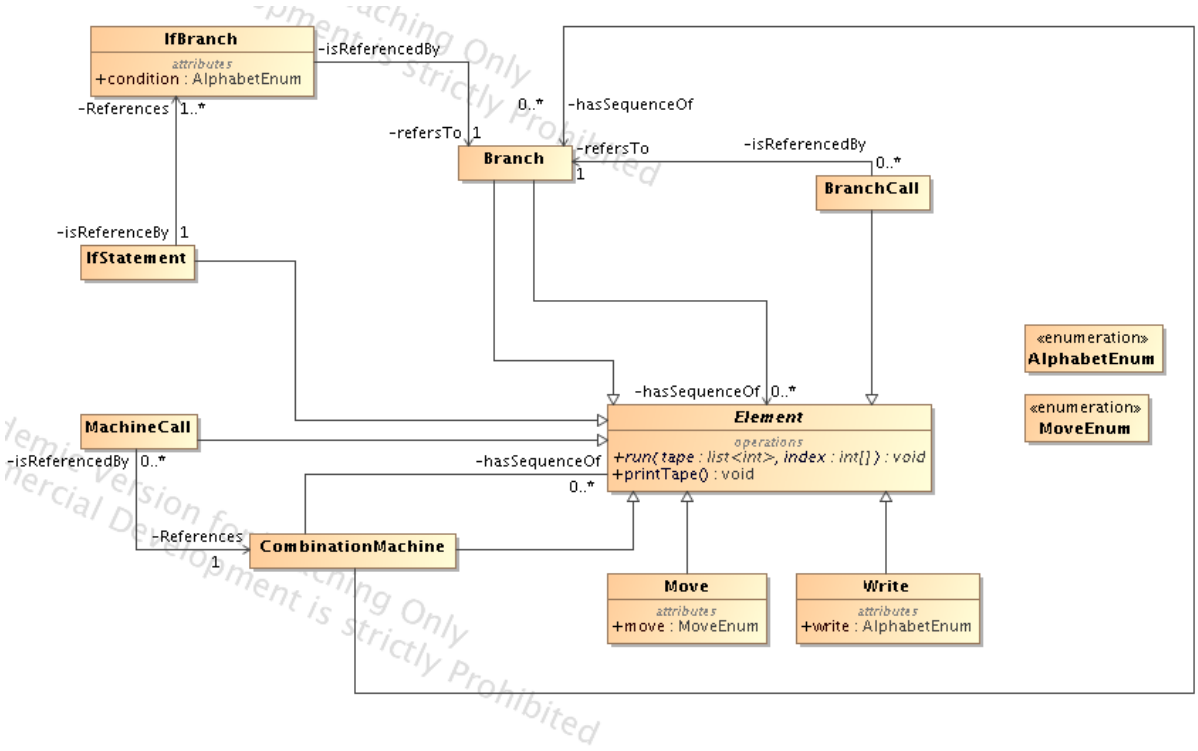

Figure 1: UML class diagram of the structure

- CombinationMachine

  CombinationMachine is a root concept which extends the Element concept. It contains a sequence of zero to n Element concepts and Branch concepts.

- Element

  Element is an abstract concept which serves as a way for parent concepts to contain sequences of different concepts in the same sequence. In other words, it allows parent concepts to contain a "generic list" of concepts.

- MachineCall

  MachineCall is a concept which extends Element and contains a reference to CombinationMachine. This concept acts as a way to reference a combination machine from a sequence in a different combination machine.

- Branch

  The Branch concept extends Element and contains a sequence of Element concepts. This concept works as a part of a combination machine sequence, and can be called by the BranchCall concept.

- BranchCall

  BranchCall is a concept that contains a reference to Branch. This way, a Branch can be reused many times without the need to recreate the branch each time.

- IfStatement

  IfStatement extends Element and is our implementation of the if-statement. Each IfStatment contains a list of one to n IfBranches.

- IfBranch

  The IfBranch contains a condition property of type AlphabetEnum to check against, and a reference to a single Branch to be run if the condition is true.

- Write

  Write is a concept that contains a write property of type AlphabetEnum that will be written to the tape.

- AlphabetEnum

  AlphabetEnum is an enumeration of the symbols used to read from and write to the tape in a runtime environment. The member type is set to integer, but could easily be replaced with another type if needed. The default value is 0.

- Move

  Move is a concept that contains a move property of type MoveEnum. The move property determines what direction the head of tape will move.

- MoveEnum

  The MoveEnum is an enumeration of the directions the of head of the tape can move. The MoveEnum enumeration is comprised of the string values "left" and "right". The default value is "left".

- TableMachine

  TableMachine is a root concept and is used to encapsulate a table machine.

- State

  The State concept contains a list of one to n Read concepts and TransitionFunction concepts. This is to ensure that a state always has a transition function to run, but also allows for an arbitrarily large alphabet.

- Read

  The Read concept is used to check if the symbol on the tape matches for a given TransitionFunction.

- TransitionFunction

  TransitionFunction is a concept with two children: Write and Move. It also contains a reference to the Next State which the Turing Machine will transition to after the transition function has finished.

3.2 Syntax

The Turing language allows the user to create a textual notation of a combination machine. The machines are presented in a way that makes it easy to understand what belongs where. Figure 2 shows the layout of a combination machine in the editor. Child concepts are indented from the parent, which makes it easier to see where the nodes are in the Abstract Syntax Tree. Keywords are marked in dark blue, while combination machines and references to them are marked in magenta, and branches and references to them are marked in dark magenta. The language contains two predefined combination machines, L and R. These combination machines move the head of the Turing machine left or right respectively for each "1" symbol they read on the tape. These machines can be accessed by referencing in the editor.

```
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 2: Textual representation of a combination machine

3.3 Semantics

The semantics of a modelling language describes what the language means in terms of its behavior, static properties or translation to another language[1]. In MPS we have described the semantics of the Turing language by defining with Java code the behavior for each concept. Since the concepts maintain properties and relationships between themselves, methods can be invoked for its children and references.

Only the CombinationMachine concept contains an intention, which executes the behavior for the concept. This restricts the user to only run complete machines and not branches, IfStatements, etc.

Most concepts inherit from the Element concept, which allowed us to have common interface the concepts. The Element behavior contains an abstract virtual method run() which is overridden by each concept that extends Element. Element also contains another method body() which is used to run the run() method as well as the printTape() method. This is used to easily get the status of the tape after each operation. Interpretation starts

from the CombinationMachine behavior where the Tape and Index of the tape head are created. The tape head and tape variables are of the type array because Java passes arrays by reference. This means that when a child that receives the tape and tape index as parameters, they can change the values in the arrays without having to return them for the update to take effect. The tape variable and tape index are sent as arguments into CombinationMachine's body method. This method is defined in the Element behavior and calls the run() method which is overridden, and the printTape() method afterwards. The functionality of each concept is implemented in their run() methods. As a result, each operation from a concept is followed by the tape content being printed.

3.4 Pros and cons

Some positive aspects of this project were:

- The structure

  The structure turned out quite well and it is capable of encapsulating Combination Turing Machines in a satisfying manner.

- The editor

  The editor looks clean and orderly. Although some improvements could be made to make it easier to insert certain concepts (like enumerations), the end result is good overall.

- The behavior

  The behavior of the language is invoked through an intention for CombinationMachine. This means a user can press ALT+ENTER when a combination machine is highlighted in the editor, and select "Run" to effectively run the combination machine. One drawback is the format of the output. Since System.out.println prints to the System, which in this case is MPS, the output is invisible to the user unless MPS is run through a terminal. Hence MPS must be initialized through a terminal in order to display output to the user.

There were some negative aspects in the project as well. As mentioned, this project was a continuation of a previous project which had revolved around implementing table machines. During development of the combination machine language, some parts of the existing implementation had to be removed due to errors.

4 Discussion

Our implementation of the Turing language differs greatly from that of the previous group. We have significantly increased the functionality by including concepts that allows branching and recursion.

We chose to represent the Turing language via a textual nodal representation that is based on the the finite state representation. We chose this because, compared to the table format, it requires less input by the user and it is easier to understand the flow of the machine and its operations.

We have implemented several Turing machines of varying complexity to test the correctness of our interpreter. All machines run correctly and produce the expected output with the exception of the Greatest Common Divisor (GCD) machine. This was revealed to not be a flaw within our interpreter but the Turing machine figure we based it on. Given an accurate model, the GCD machine should work as well.

In our implementation of the Turing language, we chose to use executional transformation in order to execute the combination machines. This in turn resulted in an interpreter for the language. An alternative to this would be to apply translational transformations in order to translate the model into Java code. This would be done through the generator.

As mentioned in section 3.3, we passed the index of the Tape as an array. This is because Java passes by value on primitive types. For example, if we had chosen to compile to C code we could use pointers to fix this issue. Despite this, we continued to use Java since MPS is built on Java and the previous group had chosen to use Java as well. Changing compile language would also be difficult and at the very least time consuming.

In the Move behavior the move property is treated as an object rather than a string for some unknown reason, which is the reason why the length of the strings "left" and "right" were used to distinguish which direction the tape head should move.

One way to view output from behavior is through the message info function provided by MPS, in which case the output would be shown in the MPS console. The problem with this approach is that the message info function did not print variables properly. Several variables such as alphabet enumeration as well as the tape and tape index are represented as integers or integer arrays, and were therefore unable to be printed. Casting these variables to string proved unsuccessful as they represented the object identifier instead of the string value.

There was an attempt to create a user interface for the output of the interpreter, but due to time constraints, this was not completed. We followed a guide for creating an interpreter in MPS [4], and started implementing functionality for a JFrame which was supposed to show the user the output log from the interpreter. The PreviewFactory class was used to create the JPanel, and some JFrame functionality lies in the interpret method in the CombinationMachine bahavior and has been commented out.

As a result of time constrains, there are still a few things that could be implemented and improved upon. We could have designed a better user interface, incorporated ScopeProvider, and we could have implemented other representations of Turing Machines in the editor such as the table or node variants.

5 Plan and reflection

During this project period, we planned work mostly over Facebook Messenger and worked together at school. In exception of this, no formal planning was done in advance of working, and the amount of working hours were not counted during working. If we had planned in advance it would possibly have been easier to work more efficiently.

We had periodic meetings with our supervisor where we received feedback on our work and were pointed in a direction of what to work on next and what could be improved upon. These meetings proved to be very useful and we got a lot of help and information.

At the start of the project we tried to build upon what the previous group had created but after some time we decided start the project over since our goal was to create a different representation of Turing Machine and as a result we lost some time at the beginning of the project period.

To conclude, the project turned out mostly successful. The Turing language contains a good structure and an easily usable editor, as well as the option to run a combination machine through an interpreter. One key feature which is currently missing is an interface for the user where the output of the interpreter is displayed.

6 References

[1] Tony Clark, Paul Sammut, and James Willans. "Applied Metamodelling: A Foundation for Language Driven Development (Second Edition)". In: (2008).

[2] Liesbeth De Mol. "Turing Machines". In: (Sept. 2018). URL: https://plato.stanford.edu/entries/turing-machine/.

[3] Robert Mullins. "What is a Turing Machine?" In: (2012). URL: https://www.cl.cam.ac.uk/projects/raspberrypi/tutorials/turing-machine/one.html.

[4] Vaclav Pech. "Building an interpreter cookbook". In: (Feb. 2016). URL: https://confluence.jetbrains.com/display/MPSD33/Building+an+interpreter+cookbook.