Modern programming language translation to the theoretical model of Minsky Machine (Counter Machine).

(Tłumaczenie współczesnego języka programowania
do Maszyny Minsky'ego.)

Jadwiga Pokorska

Praca magisterska

Promotor: dr Jakub Michaliszyn

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

9 maja 2018

Abstract

Counter Machine (a machine with finite state set, two counters and input/output tape) is able to express any computations done by modern programming languages. This is the well-known theorem, just like computations performed using Turing Machine and means that anything written in a modern programming language is possible to translate into the theoretical model of Minsky Machine (as well as into Turing Machine).

My goal is to build automatic translation from a modern programming language (C++) to Minsky Machine.

TODO: polishabstract

Contents

Chapter 1

Introduction

TODO: Co to są maszyny Turinga, Mińskiego, Brainfuck, Turning-zupełność, hipoteza Churcha

TODO: O czym jest ta praca - lista kontrybucji

TODO: Po co jest ta praca — „proof of concept”; „walory dydaktyczne”, ...

TODO: „Related work”

TODO: Plan pracy

...

Chapter 2

Preliminaries

2.1 Brainfuck

Brainfuck is a programming language with only 8 statements and execution of any program is done using a finite sequence of memory cells. Statements operate on these cells using data pointer — initially the pointer is set on the leftmost cell in the sequence.

Statements:

- `>` moves the data pointer one cell to the right,
- `<` moves the data pointer one cell to the left,
- `+` increase by 1 the value held in the cell under the data pointer,
- `-` decrease by 1 the value held in the cell under the data pointer,
- `.` print to stdout the character that is under the data pointer,
- `,` read from stdin a character and write it to the cell under the data pointer,
- [beginning of a loop with condition checking whether value under the data pointer is zero. If it is then execution jumps to the matching],
-] closing symbol of a loop — execution jumps to the beginning of the loop (matching [) and then check for zero value under the data pointer is performed.

It is allowed to use any other characters within the code, but anything else than the 8 listed above are ignored — it is useful for creating comments in the code.

An example code printing "Hello World!":

```
++++++++++
```

```
[
>+++++++>++++++++++>+++>+<<<<-
] We set up the values in few cells for future use.
>++.               prints 'H'
>+.                prints 'e'
+++++++.           prints 'l'
.                  prints 'l'
+++.               prints 'o'
>++.               prints space
<<+++++++++++++++. prints 'W'
>.                 prints 'o'
+++.               prints 'r'
------.            prints 'l'
--------.          prints 'd'
>+.                prints '!'
>.                 prints newline character
```

2.2 Turing Machine

2.2.1 Instructions

Turing Machine consists of a finite number of states, changes between them and potentially infinite tape, on which it is able to write and read symbols. It has no code sequence, a program is a set of state changes and definition of an initial state. There is special final state `END` that does not need to be defined to be used.

There is a predefined set of symbols used on the tape which is whole ASCII character set extended with the same amount of additional (non-ASCII) symbols. Regular ASCII characters have 7 bits (numbers 0-127), so tape symbols have 8 bits allowing to hold regular ASCII and special characters (numbers 128-255).

There are few special definitions (so far using regular ASCII, but it will be changed to use special characters instead):

- `BLANK` which represents empty field on the tape
- `*` (ALL) defines all characters (both ASCII and special)
- `#` (NOTHING) means no character which is used to say that we do not want to write anything on tape during this state change
- `&` (NON-ZERO) defines all characters except 0
- `0` (ZERO) represents zero that is used for conditional jumps (jump zero)

- `>` (NEXT-CHAR) allows writing on the tape next character (increased by one), i.e. writes `g` if on the tape was `f`
- `<` (PREV-CHAR) similarly as above but writes the previous character, i.e. writes `e` when seen `f`

Defining initial state:

```
START: state_name
```

Each state change looks as follows:

```
state_name symbol(s) -> target_state head_move symbol_to_write
```

`symbol(s)` is ASCII character including special definitions (currently all special definitions use regular ASCII so that it is easy to see in a standard text file), in future it will allow getting non-ASCII special characters as well.

`head move` is one of `L`, `R` or `-` which steer the head to go left, right or stay in place, respectively.

`symbol to write` is any (currently ASCII) symbol. Currently, there is no mechanism to prevent writing any symbol from special definitions, but it will cause an undefined behaviour of the machine. The only symbol from special definitions that is allowed to appear as `symbol_to_write` is `#` (NOTHING) meaning that symbol on the tape should not change.

2.2.2 Extensions of the theoretical model

Standard input/output handling is done by allowing to read or write single ASCII character from/to stdin/stdout. It is possible to add additional reading or writing before moving the head. It is done by modifying change symbol `->` in the state change definition.

- Reading is done with `->*`, i.e. `state1 A ->* state2 R NOTHING` which means when seen symbol `A` in state `state1` we read from stdin one character, overwrite `A` to read value and move head one field right.
- Writing is done similarly with `->^`, i.e. `state1 A ->^state2 R NOTHING` which will print symbol `A` on stdout and move the head to the right.

2.2.3 Example

A program that reads a letter from stdin, writes this letter into stdout and if the written letter was `B` or `b` then prints `.` at the end as well, otherwise finishes.

```
START: s1
s1 ALL ->* s2 - NOTHING
s2 ALL ->^ s3 - NEXT_CHAR
s3 ALL -> s4 - NOTHING
s4 b ->^ s5 R NOTHING
s4 B ->^ s5 R NOTHING
s5 ALL -> s6 - .
s6 ALL ->^ s7 - NOTHING
s7 ALL -> END - NOTHING
```

Note: If there is no state change defined for given configuration (nothing matches) then it is assumed that machine gets to `END` state. Because of this in the above example, last instruction is not necessary.

2.3 2 Stack Pushdown Automaton

2 Stack Pushdown Automaton consists of two stacks, input tape and definition of states and transitions between them. Each transition looks as follows:

```
state_name left_pattern right_pattern ->
    target_state left_stack_items right_stack_items
```

Explanation:

- `left_pattern` is symbol or pattern that should be matched for the symbol at the top of the left stack
- `right_pattern` is the same as above, but for right stack
- `left_stack_items` are the list of items that should be pushed to the left stack before moving to `target_state`. It might be a single letter `"a"`, a sequence of letters `"abc"` or sequence of letters and references, i.e. `("a" + ORIG_LEFT + "b")` where `ORIG_LEFT` means the letter we read from the left stack (the one matched in `left_pattern`). Note: If `+` is used it is required to put the whole sequence in parenthesis
- `right_stack_items` are the same as above, but for items to be pushed into right stack

Special references and definitions in transitions:

- `ORIG_LEFT` is the letter taken from the left stack
- `ORIG_RIGHT` is the letter taken from right stack

- `INPUT_CHAR` is the letter taken from input tape (only in input transition type - see section below)
- `NOTHING` may be used as `left_stack_items` or `right_stack_items` and means that nothing is pushed into left/right stack
- `END` is special state name into which the transition is made if no other transition is specified
- `$` is the symbol of the empty stack

2.3.1 Extensions of the theoretical model

Input/Output handling is done similarly to Turing Machine - We change `->` in transition to `->*` or `->^`.

When defining input transition it is allowed to use `INPUT_CHAR` in any items to be pushed into left/right stack. Here is an example that takes a character from input tape and pushes it into the left stack (and ignores symbols taken from both stacks).

```
state1 ALL ALL ->* state2 INPUT_CHAR NOTHING
```

When defining output transition we **must** specify what character is printed with adding `Output: <letter>` at the very back of transition definition. An example that prints letter taken from the left stack (and ignores what was taken from right stack):

```
state1 ALL ALL ->^ state2 NOTHING NOTHING Output: ORIG_LEFT
```

An example that prints letter "a" (and ignores what was taken from stacks):

```
state1 ALL ALL ->^ state2 NOTHING NOTHING Output: "a"
```

Important note: Order of defining transition matters. If patterns do not match a distinct set of letters then the transition that appeared first is applied.

2.3.2 Example

An example (equivalent to the example from Turing Machine):

```
START: init_state
init_state $ $ -> s1 BLANK NOTHING
s1 ALL ALL ->* s2 INPUT_CHAR ORIG_RIGHT
```

```
s2 ALL ALL ->^ s3 ORIG_LEFT ORIG_RIGHT Output: ORIG_LEFT
s3 b $ -> s4 (ORIG_LEFT + BLANK) $
s3 b ALL -> s4 (ORIG_LEFT + ORIG_RIGHT) NOTHING
s3 B $ -> s4 (ORIG_LEFT + BLANK) $
s3 B ALL -> s4 (ORIG_LEFT + ORIG_RIGHT) NOTHING
s4 ALL ALL -> s5 "." ORIG_RIGHT
s5 ALL ALL ->^ s6 ORIG_LEFT ORIG_RIGHT Output: ORIG_LEFT
s6 ALL ALL -> END ORIG_LEFT ORIG_RIGHT
```

2.4 Counter Machine (4 counters)

Counter Machine consists of 4 counters each holding non-negative integer, a finite number of states and transitions between them. Each transition looks as follows:

```
state_name (pattern pattern pattern pattern) ->
    target_state (number number number number)
```

Where:

- `state_name` is the state in which counter machine needs to be within for this transition to be applied,
- `pattern` is one of values `0`, `1` or `_` meaning empty counter, non-empty counter and any counter state and defines what is expected state of the given counter - the transition may be applied only when all counter states are matched (Notice that symbol `_` is matched with any state of the counter),
- `target state` is the state in which machine will be after applying the transition,
- `number` is an integer from the range `[-1, MAX_INT]` specifying what should be added to given counter - it is allowed to decrease counter by 1 only, but it is possible to increase it by any number that can be stored in regular integer type.

Note: If there are many transitions that may be applied in given state matching all counters **the first one** is applied. It means that order of defining transitions matters.

It is required to give an initial state of the machine with the following statement:

```
START: state_name
```

2.4.1 Extensions of the theoretical model

Input/Output operations fit in the schema of using counters - input and output are additional counters which transitions may use in a similar way as counters are used.

Input transition is defined as follows:

```
state_name (counters) input pattern ->*
    target_state (numbers) input_operation
```

Where:

- `input_pattern` is one of `0`, `1` or `_` (same as counter pattern) and specifies what should be the state of input counter for this transition to be applied,
- `input_operation` specifies what action should be performed on input counter and is one of `LOAD`, `-1` or `NOOP` meaning respectively loading a character from stdin into the input counter, decrease input counter by `1` and leaving input counter untouched.

This transition reads from stdin into input counter:

```
state1 (_ _ _ _) _ ->* state2 (0 0 0 0) LOAD
```

These transitions read the value from input counter and store it in first counter:

```
state1 (_ _ _ _) 1 ->* state1 (1 0 0 0) -1
state1 (_ _ _ _) 0 ->* state2 (0 0 0 0) NOOP
```

Note: It is assumed that transition may just decrease the input counter and is not allowed to increase its value directly.

Output transition is defined as follows:

```
state_name (counters) ->^ target_state (numbers) Output: output_operation
```

Where:

- `output_operation` specifies what should be performed on output counter and may be one of `FLUSH` or non-negative number, meaning respectively pushing counter to stdout and modifying the value stored in the counter by the given number.

These transitions print character stored in first counter:

```
state1 (1 _ _ _) ->^ state1 (-1 0 0 0) Output: 1
state1 (0 _ _ _) ->^ state2 (0 0 0 0) Output: FLUSH
```

Note: It is assumed that transition may just increase the output counter and is not allowed to decrease its value directly.

2.4.2 Example

Code that reads a character from stdin doubles its ASCII value and prints the result.

```
START: s1
s1 (_ _ _ _) _ ->* s2 (0 0 0 0) LOAD
s2 (_ _ _ _) 1 ->* s2 (1 0 0 0) -1
s2 (_ _ _ _) 0 ->* s3 (0 0 0 0) NOOP
s3 (1 _ _ _) -> s3 (-1 2 0 0)
s3 (0 _ _ _) -> s4 (0 0 0 0)
s4 (_ 1 _ _) ->^ s4 (0 -1 0 0) Output: 1
s4 (_ 0 _ _) ->^ END (0 0 0 0) Output: FLUSH
```

2.5 Counter Machine (2 counters)

Counter Machine with 2 counters has the same definition as Counter Machine with 4 counters, but it is allowed to use only 2 counters, so transitions become of the form:

```
state_name (pattern pattern) -> target_state (number number)
```

Input/Output is handled the same way it is handled in Counter Machine with 4 counters.

Chapter 3

Theoretical underpinnings

TODO: O tym, jak te redukcje działają

Chapter 4

Implementation

4.1 Ogólny wstęp

TODO: Uzasadnienie doboru narzędzi

TODO: Najciekawsze wyzwania

TODO: Wszystko, co było ciekawe

4.2 Szczegóły techniczne

TODO: Co jest gdzie, jak to się uruchamia, itd. / skrócona instrukcja użytkownika

TODO: Ograniczenia implementacji

4.3 Przykłady

Dużo przykładów...

4.4 Ewaluacja

Porównanie czasów

Tabelka

4.5 Optymalizacje

Chapter 5

Podsumowanie, wnioski, możliwości kontynuowania

Appendix A

Szczegółowy sposób instalacji i konfiguracji lub dostępu do działającego systemu oraz podręcznik użytkownika systemu.