Modern programming language translation to 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

2 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 well known theorem, just like computations performed using Turing Machine and means that anything written in modern programming language is possible to translate into theoretical model of Minsky Machine (as well as into Turing Machine).

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

TODO: polishabstract

Contents

Chapter 1

Introduction

...

Chapter 2

Specifications of languages

2.1 Brainfuck

TODO.

2.2 Turing Machine

2.2.1 Instructions

Turing Machine consists of 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, program is a set of state changes and definition of initial state. There is special final state `END` that does not need to be defined to be used.

There is 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 condotional jumps (jump zero)

- `>` (NEXT-CHAR) allows to write 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 previous caracted, 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 standard text file), in future it will allow to get 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 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 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 head. It is done by modifying change symbol `->` in 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 head right.

2.2.3 Example

Program that reads letter from stdin, writes this letter and next one into stdout and second 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 symbol at the top of left stack
- `right pattern` is the same as above, but for right stack
- `left stack items` is list of items that should be pushed to left stack before moving to `target state`. It might be single letter `"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 left stack (the one matched in `left pattern`). Note: If `+` is used it is required to put whole sequence in parenthesis
- `right stack items` is 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 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 where transition is made if no other transition is specified
- `$` is symbol of empty stack

2.3.1 Extensions of theoretical model

Input/Ouput 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 example that takes character from input tape and pushes it into 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. Example that prints letter taken from left stack (and ignores what was taken from right stack):

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

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 distinct set of letters then the transition that appeared first is applied.

2.3.2 Example

Example (equivalent to 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, 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`, texttt1 or texttt_ meaning empty counter, non-empty counter and any counter state and defines what is expected state of 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 integer from 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 initial state of the machine with following statement:

```
START: <state name>
```

2.4.1 Extensions of theoretical model

Input/Output operations fit in the schema of using counters - input and output are additional counters which transitions may use in 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 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 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 value stored in the counter by 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 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 like 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/Ouput is handled the same way it is handled in Counter Machine with 4 counters.

2.6 Minsky Machine - OLD

2.6.1 Instructions

Minsky Machine (currently) consists of finite number of counters each holding one non-negative integer and sequence of instructions. Each instruction might be prefixed by a label which will be used for code execution and allow to jump into desired code parts. (The idea is similar to RAM machine.)

Each of the instructions is one of the following ones:

- `INC <number>` increases given counter
- `DEC <number>` decreases given counter
- `PRINT <number>` prints state of given counter to standard output
- `READ <number>` reads number from standard input into given counter
- `JZ <number> <label>` if value of given counter is 0 then execution will continue from place in code with given label

2.6.2 Example

Program that reads number n from stdin and writes $2 \cdot n$ to stdout.

```
      READ 1
loop: JZ 1 end
      DEC 1
      INC 2
      INC 2
      JZ 3 loop
end:  PRINT 2
```