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

21 marca 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 Automata

Not yet defined.

2.4 Minsky Machine

2.4.1 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

2.4.2 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
```