Turing completeness of Mersenne Twister

Krzysztof Szewczyk

2019

1 Introduction

In this paper, I will demonstrate Mersenne Twister-generated programs being capable of interpreting a Turing-complete language. Mersenne Twister is so far the most widespread PRNG[1] in use as of 2019. Mersenne Twister will be described as a component of Seed esoteric programming language.

2 Seed programming language

Seed is a language based on arbitrary seeds. Programs exclusively contain two instructions, that is, length and random seed separated by a space. To execute a Seed program, the seed is fed into a Mersenne Twister random number generator, and the randomness obtained is transformed into a string of length bytes, which will be executed by a Befunge-98 interpreter (or compiler).[2].

An illustrational Seed program looks as follows and will generate 780 bytes long valid Befunge-98 code.

```
780 983247832
```

Since standard Befunge is considered to be a finite state machine, it is, strictly speaking, not Turing-complete; thus, Seed cannot be Turing-complete either - states Esolang Wiki[3]. Befunge-98 (the variant used by Seed) is compliant to Funge-98 standard. Wikipedia says:

The Befunge-93 specification restricts each valid program to a grid of 80 instructions horizontally by 25 instructions vertically. Program execution which exceeds these limits "wraps around" to a corresponding point on the other side of the grid; a Befunge program is in this manner topologically equivalent to a torus. Since a Befunge-93 program can only have a single stack and its storage array is bounded, the Befunge-93 language is not Turing-complete (however, it has been shown that

[1] Pseudorandom number generator
[2] https://esolangs.org/wiki/Seed - accessed 16.06.2019
[3] Before correction coerced by existence of this document

Befunge-93 is Turing-complete with unbounded stack word size). The later Funge-98 specification provides Turing completeness by removing the size restrictions on the program; rather than wrapping around at a fixed limit, the movement of a Funge-98 instruction pointer follows a model dubbed "Lahey-space" after its originator, Chris Lahey. In this model, the grid behaves like a torus of finite size concerning wrapping, while still allowing itself to be extended indefinitely.[4]

This means, Befunge-98 **is** Turing complete, but Seed might not be, because we may not be able to generate **every** imaginable Befunge-98, because Mersenne Twister has a period of $2^{19937} - 1$, but whether it was able to generate Befunge-98 program being interpreter of another Turing complete language, the proof would have been completed (as implied by thesis of simulation - *If an interpreter for A can be implemented in B, then B can solve at least as many problems as A can.*).a

3 ByteByteJump

ByteByteJump is a remarkably simplistic One Instruction Set Computer (OISC). Its sole instruction copies a byte from the given memory location to another and then performs an unconditional branch towards a point. There are two reasonable ways to prove it's Turing-complete.

ByteByteJump is a variation of BitBitJump, that is operating on bits, not bytes, therefore proving BitBitJump is Turing complete can prove Turing-completeness of ByteByteJump. Using BitBitJump assembler and standard library[5], it's possible to create a Brainfuck interpreter:

```
    # BitBitJump brainfuck (DBFI) interpreter by O. Mazonka
    Z0:0 Z1:0 start
    .include lib.bbj
    :mem:0 0 0
    mem mem
    ip_start:mem ip:mem ip_end:0
    mp_start:0 mp:0 mp_end:0
    x:0 y:0 m1:-1
    SPACE:32 MINUS:45 PLUS:43
    TICK:39 EOL:10 Excl:33
    LEFT:60 RIGHT:62 LB:91
    RB:93 DOT:46 COMMA:44
start:    .copy ZERO x
    .in x
    .ifeq x ZERO initgl chkex
chkex:    .ifeq x Excl initgl storei
storei:    .toref x ip
    .add ip BASE ip
    0 0 start
initgl:    .copy ip ip_end
    .copy ip mp_start
```

[4] https://en.wikipedia.org/wiki/Befunge - accessed 16.06.2019

[5] http://mazonka.com/bbj/bbjasm.cpp and http://mazonka.com/bbj/lib.bbj - accessed 16.06.2019

```
    .copy ip mp
    .copy ip mp_end
    .add mp_end BASE mp_end
    .copy ip_start ip
loop:    0 0
    .deref ip x
    .ifeq x PLUS plus chk_ms
plus:    .plus
    0 0 next_ip
chk_ms: .ifeq x MINUS minus chk_lt
minus:    .minus
    0 0 next_ip
chk_lt:    .ifeq x LEFT left chk_rt
left:    .left
    0 0 next_ip
chk_rt: .ifeq x RIGHT right chk_dt
right:    .right
    0 0 next_ip
chk_dt: .ifeq x DOT dot chk_cm
dot:    .deref mp x
    .out x
    0 0 next_ip
chk_cm: .ifeq x COMMA comma chk_lb
comma:    .copy ZERO x
    .in x
    .toref x mp
    0 0 next_ip
chk_lb: .ifeq x LB lb chk_rb
lb:    .lb
    0 0 next_ip
chk_rb: .ifeq x RB rb next_ip
rb:    .rb
    0 0 next_ip
next_ip: 0 0
    .add ip BASE ip
    .ifeq ip ip_end exit loop
exit:    0 0 -1
    .def plus : mp x
    .deref mp x
    .inc x
    .toref x mp
    .end
    .def minus : mp x
    .deref mp x
    .dec x
    .toref x mp
    .end
    .def right : mp BASE mp_end x ZERO
```

```
    .add mp BASE mp
    .iflt mp mp_end ret incend
incend: .add mp_end BASE mp_end
    .copy ZERO x
    .toref x mp
ret:    0 0
    .end
    .def left : mp BASE
    .sub mp BASE mp
    .end
    .def lb : ip mp x y ZERO ONE BASE LB RB
    .deref mp x
    .ifeq x ZERO gort ret
gort:    .copy ONE y
loop:    .add ip BASE ip
    .deref ip cmd
    .ifeq cmd LB incy chk_rb
chk_rb: .ifeq cmd RB decy loop

incy:    .inc y
    0 0 loop
decy:    .dec y
    .iflt ZERO y loop ret
    cmd:0 0
ret:    0 0
    .end
    .def rb : ip mp x y ZERO ONE BASE LB RB
    .deref mp x
    .ifeq x ZERO ret golt
golt:    .copy ONE y
loop:    .sub ip BASE ip
    .deref ip cmd
    .ifeq cmd RB incy chk_lb
chk_lb: .ifeq cmd LB decy loop
incy:    .inc y
    0 0 loop
decy:    .dec y
    .iflt ZERO y loop ret

    cmd:0 0
ret:    0 0
    .end
    .def dump : ip_start x y BASE ip_end mp_start mp_end SPACE ip mp TICK EOL
    .copy ip_start y
dumpi:    .deref y x
    .out x
    .ifeq y ip outi noi
outi:    .out TICK
```

```
noi:    .add y BASE y
    .ifeq y ip_end dumpmst dumpi
dumpmst: 0 0
    .copy mp_start y
dumpm:    .deref y x
    .out SPACE
    .ifeq y mp outm nom
outm:    .out TICK
nom:    .prn x
    .add y BASE y
    .ifeq y mp_end ret dumpm
ret:     .out EOL
    .end
```

To prove that Brainfuck is Turing-complete (we don't actually have to, because this has been proven before, but for the sake of consistency, the proof will be included), we will use the same rule. Daniel B. Cristofani wrote a generic Turing machine emulator, that I may use as a part of my proof:

```
+++>++>>>+[>>,[>+++++<[[->]<<]<[>]>]>-[<<+++++>>-[<<---->>-[->]<]]<[
<-<[<]+<+[>]<<+>->>>]<]<[<]>[-[>++++++<-]>[<+>-]+<<<+++>+>[-[<<+>->-
[<<[-]>>-[<<++>+>-[<<-->->>+++<-[<<+>+>>--<-[<<->->-[<<++++>+>>+<-[>
-<-[<<->->-[<<->>-[<<+++>>>-<-[<<----->>>++<-[<<++>>>+<-[>[-]<-[<<->>
>+++<-[<<->>>--<-[<<++++>+>>+<-[<<[-]>->>++<-[<<+++++>+>>--<-[<->>++
<[<<->>-]]]]]]]]]]]]]]]]]]]]]]]]<[->>[<<+>>-]<<<[>>>+<<<-]<[>>>+<<<-]]
>>]>[-[---[-<]]>]>[+++[<+++++>--]>]+<++[[>+++++<-]<]>>[-.>]
```

This means, Brainfuck and BitBitJump are Turing-complete and may prove that ByteByteJump is Turing-complete too. One could argue here, how does the property of BitBitJump coerce property of ByteByteJump (albeit the only difference between them is copy size). This statement might be completely right, so I've prepared another way to prove this using Subleq as a part of my proof. It's possible to simulate one Subleq instruction tick in ByteByteJump (the solution has been partly presented by Esolang wiki). Suppose we have the following values stored in memory:

Address	Value
000800..00087F	01
000880..0008FF	02
01XXYY	XX
02XXYY	YY
03XXYY	XX-YY

Then the following ByteByteJump program (using 3-byte addresses) will take the byte value at address 100h, subtract the byte value at address 200h, store the resulting byte value at address 300h, and jump to address 400h if the result was negative (¿= 80h).

```
Big-endian version           | Little-endian version
-----------------------------+-----------------------------
000000: 000100 000013 000009 | 000000: 000100 000013 000009
000009: 000200 000014 000012 | 000009: 000200 000012 000012
000012: 030000 000300 00001B | 000012: 030000 000300 00001B
00001B: 000300 000026 000024 | 00001B: 000300 000024 000024
000024: 000800 00002D 00002D | 000024: 000800 00002F 00002D
00002D: 003F36 000035 000000 | 00002D: 003F36 000033 000000
000036: 000000 000000 000400 | 000036: 000000 000000 000400
00003F: ...... ...... ...... | 00003F: ...... ...... ......
```

Below is the previous ByteByteJump example rewritten in ByteByte/Jump machine code.

```
Big-endian version           | Little-endian version
-----------------------------+-----------------------------
000000: 000100 00000D        | 000000: 000100 00000D
000006: 000200 00000E        | 000006: 000200 00000C
00000C: 030000 800300 000017 | 00000C: 030000 800300 000015
000015: 000800 00001B        | 000015: 000800 00001D
00001B: 002724 000023        | 00001B: 002724 000021
000021: 800000               | 000021: 800000
000024: 800400               | 000024: 800400
000027: ......               | 000027: ......
```

The ByteByteJump version takes up 63 bytes, while the ByteByte/Jump version takes up 39 bytes. At address 00000C in the ByteByte/Jump program, notice the use of multiple (in this case 2) destinations for the move instruction. This proves that ByteByteJump can compute as much as Subleq can, therefore whether Subleq is Turing-complete, ByteByteJump is too.

This Subleq assembly [6] program is emulating Brainfuck interpreter:

```
    top:top top sqmain

_interpret:
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; bp 0
    bp; sp bp
    c2 sp
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t1 0
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t2 0
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t3 0
```

[6] http://mazonka.com/subleq/sqasm.cpp - accessed 16.06.2019

```
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t4 0
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t5 0

    t1; t2; bp t1; c1 t1; t1 t2
    t1; t3; bp t1; c2 t1; t1 t3
    ?+23; ?+21; ?+24; t3 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; t2 Z; Z 0; Z

    t1; t2; bp t1; c4 t1; t1 t2
    ?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; c3 Z; Z 0; Z
l1:
    t2; t1; bp t2; c5 t2; t2 t1
    t2; t3; ?+11; t1 Z; Z ?+4; Z; 0 t2; t2 t3
    t1; t2; bp t1; c4 t1; t1 t2
    t1; t4; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t4
    t2; t1; t3 t2; t4 t2; t2 t1
    t2; t4; ?+11; t1 Z; Z ?+4; Z; 0 t2; t2 t4
    t1; t4 Z; Z t1 ?+3; Z Z ?+9; Z; t4 t1; t4 t1
    Z t1 l3
    t2; t4; bp t2; c5 t2; t2 t4
    t2; t3; ?+11; t4 Z; Z ?+4; Z; 0 t2; t2 t3
    t4; t2; bp t4; c4 t4; t4 t2
    t4; t5; ?+11; t2 Z; Z ?+4; Z; 0 t4; t4 t5
    t2; t4; t3 t2; t5 t2; t2 t4
    t2; t5; ?+11; t4 Z; Z ?+4; Z; 0 t2; t2 t5
    t4; t2; bp t4; dec t4; t4 t2
    ?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; t5 Z; Z 0; Z

    t2; t3; bp t2; dec t2; t2 t3
    t2; t5; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t5
    t3; t5 Z; Z t3; Z; c14 t3 ?+3
    t3 t3 ?+9; t3 Z ?+3; Z Z ?+3; inc t3
    Z t3 l21
    t1; t2; bp t1; c2 t1; t1 t2
    t2 Z; ?+9; Z ?+5; Z; inc 0

    Z Z l22
l21:
    t5; t2; bp t5; dec t5; t5 t2
    t5; t3; ?+11; t2 Z; Z ?+4; Z; 0 t5; t5 t3
    t2; t3 Z; Z t2; Z; c13 t2 ?+3
    t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
    Z t2 l19
    t1; t2; bp t1; c2 t1; t1 t2
```

```
    t2 Z; ?+9; Z ?+5; Z; dec 0

    Z Z l20
l19:
    t3; t5; bp t3; dec t3; t3 t5
    t3; t2; ?+11; t5 Z; Z ?+4; Z; 0 t3; t3 t2
    t5; t2 Z; Z t5; Z; c12 t5 ?+3
    t5 t5 ?+9; t5 Z ?+3; Z Z ?+3; inc t5
    Z t5 l17
    t1; t2; bp t1; c2 t1; t1 t2
    t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
    t3 Z; ?+9; Z ?+5; Z; inc 0

    Z Z l18
l17:
    t2; t3; bp t2; dec t2; t2 t3
    t2; t5; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t5
    t3; t5 Z; Z t3; Z; c11 t3 ?+3
    t3 t3 ?+9; t3 Z ?+3; Z Z ?+3; inc t3
    Z t3 l15
    t1; t2; bp t1; c2 t1; t1 t2
    t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
    t3 Z; ?+9; Z ?+5; Z; dec 0

    Z Z l16
l15:
    t5; t2; bp t5; dec t5; t5 t2
    t5; t3; ?+11; t2 Z; Z ?+4; Z; 0 t5; t5 t3
    t2; t3 Z; Z t2; Z; c10 t2 ?+3
    t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
    Z t2 l13
    t1; t2; bp t1; c2 t1; t1 t2
    t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
    t2; t1; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t1
    t1 (-1)

    Z Z l14
l13:
    t3; t5; bp t3; dec t3; t3 t5
    t3; t2; ?+11; t5 Z; Z ?+4; Z; 0 t3; t3 t2
    t5; t2 Z; Z t5; Z; c9 t5 ?+3
    t5 t5 ?+9; t5 Z ?+3; Z Z ?+3; inc t5
    Z t5 l11
    t1; (-1) t1
    t2; t3; bp t2; c2 t2; t2 t3
    t2; t4; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t4
    ?+23; ?+21; ?+24; t4 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; t1 Z; Z 0; Z
```

```
    Z Z l12
l11:
    t1; t2; bp t1; dec t1; t1 t2
    t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
    t2; t3 Z; Z t2; Z; c7 t2 ?+3
    t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
    t3; Z t2 l9
    t1; t4; bp t1; c2 t1; t1 t4
    t1; t5; ?+11; t4 Z; Z ?+4; Z; 0 t1; t1 t5
    t4; t1; ?+11; t5 Z; Z ?+4; Z; 0 t4; t4 t1
    t5; t1 Z; Z t5 ?+3; Z Z ?+9; Z; t1 t5; t1 t5
    Z t5 l9; inc t3;
l9:
    Z t3 l10
    t1; t2; bp t1; c6 t1; t1 t2
    ?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; dec Z; Z 0; Z

l4:
    t1; t2; bp t1; c6 t1; t1 t2
    t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
    t2; t3 Z; Z t2; Z; c3 t2
    Z t2 l5
    t1; t2; bp t1; c5 t1; t1 t2
    t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
    t2; t1; bp t2; c4 t2; t2 t1
    t1 Z; ?+9; Z ?+5; Z; dec 0
    t2; t4; ?+11; t1 Z; Z ?+4; Z; 0 t2; t2 t4
    t1; t2; t3 t1; t4 t1; t1 t2
    t1; t4; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t4
    t2; t1; bp t2; dec t2; t2 t1
    ?+23; ?+21; ?+24; t1 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; t4 Z; Z 0; Z

    t2; t3; bp t2; dec t2; t2 t3
    t2; t1; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t1
    t3; t1 Z; Z t3; Z; c8 t3 ?+3
    t3 t3 ?+9; t3 Z ?+3; Z Z ?+3; inc t3
    Z t3 l7
    t1; t2; bp t1; c6 t1; t1 t2
    t2 Z; ?+9; Z ?+5; Z; dec 0

    Z Z l8
l7:
    t1; t2; bp t1; dec t1; t1 t2
    t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
    t2; t3 Z; Z t2; Z; c7 t2 ?+3
```

```
    t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
    Z t2 l6
    t1; t2; bp t1; c6 t1; t1 t2
    t2 Z; ?+9; Z ?+5; Z; inc 0

l6:
l8:

    Z Z l4
l5:

l10:
l12:
l14:
l16:
l18:
l20:
l22:

l2:
    t4; t2; bp t4; c4 t4; t4 t2
    t2 Z; ?+9; Z ?+5; Z; inc 0
    Z Z l1
l3:

    ?+8; sp ?+4; t5; 0 t5; inc sp
    ?+8; sp ?+4; t4; 0 t4; inc sp
    ?+8; sp ?+4; t3; 0 t3; inc sp
    ?+8; sp ?+4; t2; 0 t2; inc sp
    ?+8; sp ?+4; t1; 0 t1; inc sp
    sp; bp sp
    ?+8; sp ?+4; bp; 0 bp; inc sp
    ?+8; sp ?+4; ?+7; 0 ?+3; Z Z 0

_main:
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; bp 0
    bp; sp bp
    c15 sp
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t1 0
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t2 0
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t3 0
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t4 0
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
```

```
    ?+6; sp ?+2; t5 0
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; t6 0

    t1; t2; bp t1; c15 t1; t1 t2
    ?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; dec Z; Z 0; Z

l23:
    t1; t2; bp t1; c15 t1; t1 t2
    t1; t3; ?+11; t2 Z; Z ?+4; Z; 0 t1; t1 t3
    t2; t3 Z; Z t2 ?+3; Z Z ?+9; Z; t3 t2; t3 t2
    Z t2 l25
    t3; (-1) t3
    t1; t4; bp t1; dec t1; t1 t4
    t1; t5; bp t1; c16 t1; t1 t5
    t1; t6; ?+11; t5 Z; Z ?+4; Z; 0 t1; t1 t6
    t5 Z; ?+9; Z ?+5; Z; inc 0
    t5; t1; t4 t5; t6 t5; t5 t1
    ?+23; ?+21; ?+24; t1 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; t3 Z; Z 0; Z

    t2; t1; bp t2; dec t2; t2 t1
    t2; t3; bp t2; c16 t2; t2 t3
    t2; t4; ?+11; t3 Z; Z ?+4; Z; 0 t2; t2 t4
    t3; t4 Z; Z t3; Z; dec t3
    t4; t2; t1 t4; t3 t4; t4 t2
    t4; t3; ?+11; t2 Z; Z ?+4; Z; 0 t4; t4 t3
    t2; t3 Z; Z t2; Z; c17 t2 ?+3
    t2 t2 ?+9; t2 Z ?+3; Z Z ?+3; inc t2
    t3; inc t3; Z t2 ?+3; Z Z l26
    t4; t1; bp t4; dec t4; t4 t1
    t4; t5; bp t4; c16 t4; t4 t5
    t4; t6; ?+11; t5 Z; Z ?+4; Z; 0 t4; t4 t6
    t5; t6 Z; Z t5; Z; dec t5
    t6; t4; t1 t6; t5 t6; t6 t4
    t6; t5; ?+11; t4 Z; Z ?+4; Z; 0 t6; t6 t5
    t4; t5 Z; Z t4; Z; c18 t4 ?+3
    t4 t4 ?+9; t4 Z ?+3; Z Z ?+3; inc t4
    Z t4 ?+3; Z Z l26; t3;
l26:
    Z t3 l27
    t1; t2; bp t1; c15 t1; t1 t2
    ?+23; ?+21; ?+24; t2 Z; Z ?+10; Z ?+8
    Z ?+11; Z; 0; c3 Z; Z 0; Z
l27:

l24:
```

```
    Z Z l23
l25:

    t1; t2; bp t1; dec t1; t1 t2
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+9; sp ?+5; t2 Z; Z 0; Z
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; ?+2 0 _interpret; . ?;
    c5 sp

    ?+8; sp ?+4; t6; 0 t6; inc sp
    ?+8; sp ?+4; t5; 0 t5; inc sp
    ?+8; sp ?+4; t4; 0 t4; inc sp
    ?+8; sp ?+4; t3; 0 t3; inc sp
    ?+8; sp ?+4; t2; 0 t2; inc sp
    ?+8; sp ?+4; t1; 0 t1; inc sp
    sp; bp sp
    ?+8; sp ?+4; bp; 0 bp; inc sp
    ?+8; sp ?+4; ?+7; 0 ?+3; Z Z 0

sqmain:
    dec sp; ?+11; sp ?+7; ?+6; sp ?+2; 0
    ?+6; sp ?+2; ?+2 0 _main; . ?; inc sp

    Z Z (-1)

. c5:-2 c3:0 c17:10 c18:13 c4:2 c2:20 c6:3 c1:4 c12:43 c9:44 c11:45 c10:46 c16:501
 (wrapped) c15:502 c13:60 c14:62 c8:91 c7:93

. t1:0 t2:0 t3:0 t4:0 t5:0 t6:0

. inc:-1 Z:0 dec:1 ax:0 bp:0 sp:-sp
```

The interpreter was built by translating the following C code [7][8]:

```
void interpret(char * input) {
    char current_char;
    int i, loop;
    unsigned char tape[16] = {0};
    unsigned char * ptr = tape;

    for (i = 0; input[i] != 0; i++) {
        current_char = input[i];
```

[7] https://gist.github.com/maxcountryman/1699708 - accessed 16.06.2019

[8] Please note that the code was modified to run smoothly on the Subleq virtual machine

```
        if (current_char == '>') {
            ++ptr;
        } else if (current_char == '<') {
            --ptr;
        } else if (current_char == '+') {
            ++*ptr;
        } else if (current_char == '-') {
            --*ptr;
        } else if (current_char == '.' ) {
            putchar(*ptr);
        } else if (current_char == ',') {
            *ptr = getchar();
        } else if (current_char == ']' && *ptr) {
            loop = 1;
            while (loop > 0) {
                current_char = input[--i];
                if (current_char == '[') {
                    loop--;
                } else if (current_char == ']') {
                    loop++;
                }
            }
        }
    }
}


int main(void) {
    char buf[500], c, r = 1;

    while(r) {
        buf[c++] = getchar();
        if(buf[c-1] == 10 || buf[c-1] == 13)
            r = 0;
    }

    interpret(buf);
}
```

It's able to hold up to 500 program bytes (just enough to simulate Cristofani's Turing machine emulator) and 16 memory cells. That may not be enough, but the size can be flexibly changed. The resulting Subleq code is quite large[9], scoring 16 kilobytes. Turing-completeness of Brainfuck has been proven before, therefore to prove Turing-completeness of Seed, it's required to create ByteByteJump interpreter in Befunge-98, and then transform it to Seed.

[9] https://pastebin.com/TEgwuLsb - accessed 16.06.2019

4 Proof

The following code ByteByteJump interpreter I wrote will be a reference for the later code.

```
a[99], b, c;

main() {
    for (; b<99; b++)
        scanf("%d",&a[b]);

    for (; !c<0; c++) {
        if (a[c]<0)
            a[c]=getchar();
        if (a[c+1]<0)
            putchar(a[c+1]);

        a[c+1]=a[c];
        c=a[c+2]
    }
}
```

If elements were listed in counter-order they are pushed, A is equal to `*(pc)` and is first, B is second and is equal to `*(pc+1)` and C is third and is equal to `*(pc+2)`, following Befunge-98 code will perform ByteByteJump instruction tick[10]:

```
v@0~$<
>:0‘!|
v‘0:\<
>!v  >’$\@
 v_$:^
 >:,0@
```

The problem here is, the registers are immutable, so the place marked with apostrophe needs to implement some kind of storing register values once again.

Finally, the Seed code looks such:

```
45 14759728162827650584261424512858419005895877927172867170454220018091585985218474762085902733083
```

And it's **4 637 bytes** big. Given all these step-by-step chaining lemmas, the final and proven thesis is, Seed language is Turing-complete, and Mersenne Twister can generate valid Befunge-98 program, that will be capable of simulating any algorithm's logic can be constructed.

[10] If anything is wrong, please contact me!

5 How did the reversing happen?

The reversal has been done step by step - first, my generator was very naive, so the program size was quite large, then I've managed to reverse Mersenne Twister in an even more efficient way, and finally, I've managed to write a pruning bruteforce program, that generated such piece of code as above. There are a couple of ways to bruteforce a program (the most common one is a naive bruteforce, a quite slow one).

6 Sources

This paper wouldn't be possible to write (or be significantly harder to) without work of Oleg Mazonka (his BitBitJump, ByteByteJump and Subleq work), Daniel B. Cristofani (for his Brainfuck universal Turing machine emulator), Esolang Wiki and Wikipedia proofreaders and content creators (for great explaining, a lot of effort and time put into the wiki), Mersenne Twister crackers preceding me (it would be significantly harder to get such score on my crack), Chris Pressey (for Befunge),