Turing completeness of Mersenne Twister

Kamila Szewczyk

2019, 2021

1 Introduction

In this paper, I will prove Mersenne Twister turing complenetess. Mersenne Twister is by far the most popular PRNG[1] in use as of 2019. Mersenne Twister will be demonstrated as component of Seed esoteric programming language.

This revision of this document differs from the other one which is generally available. It contains a few corrections and language issues.

2 Seed programming language

Seed is a language based on random data. Programs only contain two instructions: 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 converted into a string of length bytes, which will be executed by a Befunge-98 interpreter (or compiler).[2].

An example Seed which generates 780 byte-long valid Befunge-98 code snippet follows.

```
780 983247832
```

According to the Esolang wiki - *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.* Befunge-98 is compiliant 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

[1] PseudoRandom Number Generator
[2] https://esolangs.org/wiki/Seed - accessed 16.06.2019

array is bounded, the Befunge-93 language is not Turing-complete (however, it has been shown that 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 with respect to wrapping, while still allowing itself to be extended indefinitely.[3]

This means, Befunge-98 **is** Turing complete, but Seed may not be, because we aren't able to generate **all** possible befunge programs, because Mersenne Twister has a period of $2^{19937} - 1$, but, if Mersenne Twister can generate befunge program being another Turing-complete language interpreter, Seed is Turing-complete too.

3 ByteByteJump

ByteByteJump is an extremely simple One Instruction Set Computer (OISC). Its single instruction copies 1 byte from a memory location to another, and then performs an unconditional jump. There are two possible ways to prove Turing completeness of ByteByteJump.

ByteByteJump is actually a variation of BitBitJump, that is operating on bits, not bytes, therefore proving BitBitJump is Turing complete can prove Turing completness of ByteByteJump. Simulation is one way to prove Turing completeness - *If an interpreter for A can be implemented in B, then B can solve at least as many problems as A can.* Using BitBitJump assembler and standard library[4], it's possible to create 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
```

[3] https://en.wikipedia.org/wiki/Befunge - accessed 16.06.2019
[4] http://mazonka.com/bbj/bbjasm.cpp and http://mazonka.com/bbj/lib.bbj - accessed 16.06.2019

```
initgl: .copy ip ip_end
.copy ip mp_start
.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 Turing completness of Brainfuck, we will use the same rule. Daniel Cristofani wrote generic Turing machine emulator written in Brainfuck:

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

This means, Brainfuck and BitBitJump are Turing-complete, and may possibly prove that ByteByteJump is Turing-compelete too. Another method to prove Turing-completeness of ByteByteJump has been partly presented by Esolangs wiki. It's possible to simulate one SUBLEQ instruction tick in ByteByteJump. 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). Addresses which differ between the big-endian and little-endian versions are marked as bold.

```
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. Instruction words which differ between the big-endian and little-endian versions are marked as bold.

```
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, so if SUBLEQ is turing complete, ByteByteJump is too.

The following SUBLEQ assembly [5] program is a 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
```

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

```
?+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

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 from following C code [6]:

```
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];
```

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

```
        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];
    char c, r = 1;
    for(;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 pretty big[7]. Turing-completeness of Brainfuck was 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.

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

4 Proof

The following code is a ByteByteJump reference implementation written in C:

```
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 code will perform ByteByteJump instruction tick[8]:

```
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 14759728162827650584261424512858419005895877927172867170454220018091585985218474762085902733
08382625106805960994384051908698807672588439507895879458904274584488610673759294967291035323690
73598527090939875456796063044252947656841133103916443894914738745604947766408251854117733611553
00990850990369822774005176483867432996215272027868894139015703661427796383218596643499838024600
```

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

```
65526705469207771854000687726323440697970996524188895261863532132120884545889746663176353481069
79935680240791290090149417171075577498505649131916619364585867917393770995309306658514681232963
51748653781604913119842237411431221626790962751759928247159216436029135498781816145088780705956
83661859114444215229837991177129538009746262260916834297591128293975914611286552584204008881795
52979261649256152876610533168617981279499837752820315459227498068257965714456527406636705455609
43568367423856852634278745004342889968051785597832691204510364156575938067476085840495286933849
30111523198299518744607035577928208931596745818992180315322464114835086686825778874408140223240
34007117803877185190111112009676061358595435952632208157171971258305782688073583718330983714480
08987559913512056549120127899229363310257363705314682567898390346045689433922715130292374612103
95633033823486123099069458662826571247055959321928379943621757133495719495907466935865828139126
30560884886153316314520885231635153124526670384119179834142221231527301953778223587174112809062
90886132046223328553038872108648948685488002500392984258285670521772804200662659718966456688876
70764123962976484814093757725373404041684933319320964949522083970224090768751236641040408715962
74753795073454830046560959145095950925921207446560292375210823801796333887596239321887468006769
84948521469106535310244217239583470570828803144516575127575089541478487922869779458209619336688
18629369578137723222778526118402320967007156334341972358341728380660282988305399132983065625140
93855837315307576878288309079317989681042738454218976808419315765750878075196124684271652148779
25085362091308710344651493555326224269698020480102462098256214986612141204479950556249394957442
72938991601031018248427422935230882032700686694272848883560616463763763338471815730071625280265
28318382922344557070996763100739468635099643302175771697267011193485897209037417355725903667537
19975895356329798071254491342578606726886141228042235257952531167961035578401495446221035361455
92251911251788969629590777585670517302793507488111822108795511728393763085664079338676615829331
86155837300850278467843668689093691068087011047157228105003524547009141180293829599233582014457
70970530666384355236832284094858532979170175756493007083906372644947560502353750754104302433701
65758031909264984071397952981293881482392544479869133489859860114699639566779561617089469101905
45640744999756468036998978580881883135473167991118296845371856698923392214856776751663948265388
57356814611751087893422989500851252629265875041469005811478084658567757370493665254592190823074
54293368248782920259566262172445277925101602784589537602817057904193291948162808009562970003232
97875771434311924134510597121455510059391790440758628789066960212387395064111224439766822746245
03571069996651479567989030425139112053833195908709151452004833258943574098970080890242640710070
03297554083976318335685296521495235740826639772804151147807291123053024424790764106918806154830
59813832881282493316801987417483606270078165617674087424704818383812038932876146621124234091723
48680385845390436099186557158672446334565132794752214400565827293803272736638564580506051589204
63302204124540564415284465937722116774897556875744790027258847421770458552441054345596429041830
39325166031925715312874644159313315575912035447024543129526635036390453041711773525436959715026
57427679707817721788634603796757417783506559846731593877742404831352318677733816759718954567808
59338322100167998093294463760508235107869848055352454546982145805040952393877630209528468131126
20448322912407987663319745049306914142480010993518250494677632787440997943848186189775327144000
21702189807167336416774058995168777870770845307953994434957294550201432709796418079546060877975
63781195044950468159183292038174700333038378114499595619661374160798478505826254462728123747300
27090185079148589056889360511311806586580544725816247876198106128751886998044325406570156688712
80687303625216211285787804946220183255872567236414553614612979498889469153759765356621415952951
60127402720352355235679197553684532174869023372110707846052359142386881950279879397712445766312
09717694187529732570930408369359028747795329022220159904333190339104927908533379343350600156136
1851783494433943372288770941458282537803889664161030134180248397881361971773 4
```

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.

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),

7 Elegant proofs (2021)

Back in 2019, when I wrote this paper, my technology wasn't advanced enough to conduct a full-featured, convincing proof. From the current perspective, I think there are a couple of better ways to conduct the proof. One of them is implementing the Collatz function (Befunge-93 example):

```
&>:::.1-!#@_2%v
 ^        +1*3_2/
```

Obviously, this program won't work with bounded integers. A solution to this problem would be using fingerprints, or implementing arbitrary-precision numbers in Befunge (for example: Befunge-98 can shell out to Perl, so for example an I/D machine interpreter in Perl (which is around 12 bytes) can be used as a part of the proof). The Befunge code can later be compiled using techniques outlined in my PRNG cracking paper, which was hosted on my website and previously cited by Wikipedia.