Interpreter, Compiler, JIT from scratch

Jim Huang <jserv.tw@gmail.com>

Agenda

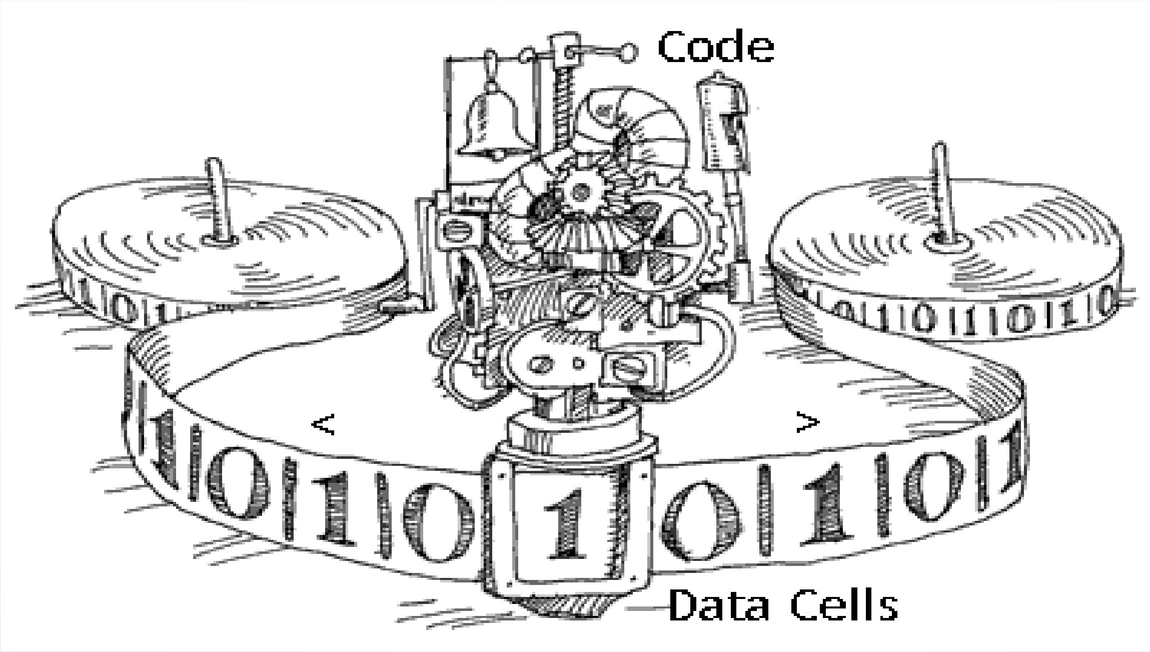
- Brainf*ck: Turing complete programming language
- Interpreter
- Compiler
- JIT Compiler

Brainf*ck

Brainf*ck Machine

- Assume we have unlimited length of buffer

Address	0	1	2	3	4	...
Value	0	0	0	0	0	...



Source: http://jonfwilkins.blogspot.tw/2011/06/happy-99th-birthday-alan-turing.html

Brainf*ck instructions

Brainfuck	C	
>	++p;	Increment the data pointer to point to the next cell.
<	--p;	Decrement the data pointer to point to the previous cell.
+	++*p;	Increment the byte value at the data pointer.
–	--*p;	Decrement the byte value at the data pointer.
.	putchar(*p);	Output the byte value at the data pointer.
,	*p = getchar();	Input one byte and store its value at the data pointer.
[	while (*p) {	If the byte value at the data pointer is zero, jump to the instruction following the matching ***]*** bracket. Otherwise, continue execution.
]	}	Unconditionally jump back to the matching ***[*** bracket.

Turing Complete

- Brianfuck has 6 opcode + Input/Output commands
- gray area for I/O (implementation dependent)
  - EOF
  - tape length
  - cell type
  - newlines
- That is enough to program!
- Extension: self-modifying Brainfuck

  https://soulsphere.org/hacks/smbf/

Brainf*ck

Address	0	1	2	3	4	...
Value	0	0	0	0	0	...

+	>	+	<	-

Brainf*ck

Address	0	1	2	3	4	...
Value	1	0	0	0	0	...

+	>	+	<	-

Brainf*ck

Address	0	1	2	3	4	...
Value	1	0	0	0	0	...

+	>	+	<	-

Brainf*ck

Address	0	1	2	3	4	...
Value	1	1	0	0	0	...

+	>	+	<	-

Brainf*ck

Address	0	1	2	3	4	...
Value	1	1	0	0	0	...

+	>	+	<	-

Brainf*ck

Address	0	1	2	3	4	...
Value	0	1	0	0	0	...

+	>	+	<	-

Brainf*ck とは

Address	0	1	2	3	4	…
Value	64	0	0	0	0	…

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

Brainf*ck

Address	0	1	2	3	4	...
Value	64	0	0	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	0	0	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	0	0	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	65	0	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	65	0	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	65	0	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	65	66	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	65	66	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	65	66	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	65	66	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck

Address	0	1	2	3	4	...
Value	64	65	66	0	0	...

.	[	>	,	]

Input

'A'	'B'	'\0'

Output

'@'		

Brainf*ck Interpreter

Initialization

- A Brainfuck program operates on a 30,000 element byte array initialized to all zeros. It starts off with an instruction pointer, that initially points to the first element in the data array or “tape.”

```
// Initialize the tape with 30,000 zeroes.
uint8_t tape [30000] = { 0 };

// Set the pointer to the left most cell of the tape.
uint8_t* ptr = tape;
```

Data Pointer

- Operators, **>** and **<** increment and decrement the data pointer.

```
case '>': ++ptr; break;
case '<': --ptr; break;
```

- One thing that could be bad is that because the interpreter is written in C and we're representing the tape as an array but we're not validating our inputs, there's potential for stack buffer overrun since we're not performing bounds checks. Again, punting and assuming well formed input to keep the code and the point more precise.

Cells pointed to Data Pointer

- **+** and **–** operators are used for incrementing and decrementing the cell pointed to by the data pointer by one.

```
case '+': ++(*ptr); break;
case '-': --(*ptr); break;
```

Input and Output

- The operators . and , provide Brainfuck’s only means of input or output, by writing the value pointed to by the instruction pointer to stdout as an ASCII value, or reading one byte from stdin as an ASCII value and writing it to the cell pointed to by the instruction pointer.

```
case '.': putchar(*ptr); break;
case ',': *ptr = getchar(); break;
```

Loop

- Looping constructs, ***[*** and ***]***.
  - ***[***: if the byte at the data pointer is zero, then instead of moving the instruction pointer forward to the next command, jump it forward to the command after the matching] command
  - ***]***: if the byte at the data pointer is nonzero, then instead of moving the instruction pointer forward to the next command, jump it back to the command after the matching [command.

Loop

```
case '[':
 if (!(*ptr)) {
   int loop = 1;
   while (loop > 0) {
     uint8_t current_char =
         input[++i];
     if (current_char == ']') {
       --loop;
     } else if (current_char == '[') {
       ++loop;
     }
   }
 }
 break;
```

```
case ']':
  if (*ptr) {
    int loop = 1;
    while (loop > 0) {
      uint8_t current_char =
          input[--i];
      if (current_char == '[') {
        --loop;
      } else if (current_char == ']') {
        ++loop;
      }
    }
  }
  break;
```

Verify the simple interpreter

- Fetch and build

```
$ git clone && cd jit-construct
$ make
$ ./interpreter samples/hello.bf
Hello World!
```

- Check interpreter.c which reads a byte and immediately performs an action based on the operator.
- Benchmark on GNU/Linux for x86_64

```
$ time ./interpreter samples/mandelbrot.b
real   2m1.297s
user   2m1.368s
sys 0m0.004s
```

Brainf*ck Compiler

[**x86_64**/x64 backend]

Generate machine code

- How about if we want to compile the Brainfuck source code to native machine code?
  - Instruction Set Architecture (ISA)
  - Application Binary Interface (ABI)
- Iterate over every character in the source file again, switching on the recognized operator.
  - print assembly instructions to stdout.
  - Doing so requires running the compiler on an input file, redirecting stdout to a file, then running the system assembler and linker on that file.

Prologue for compiled x86_64

```
const char *const prologue =
    ".text\n"
    ".globl _main\n"
    "main:\n"
    "    pushq %rbp\n"
    "    movq %rsp, %rbp\n"
    "    pushq %r12\n"        // store callee saved register
    "    subq $30008, %rsp\n" // allocate 30,008 B on stack, and realign
    "    leaq (%rsp), %rdi\n" // address of beginning of tape
    "    movl $0, %esi\n"     // fill with 0's
    "    movq $30000, %rdx\n" // length 30,000 B
    "    call memset\n"       // memset
    "    movq %rsp, %r12";
puts(prologue);
```

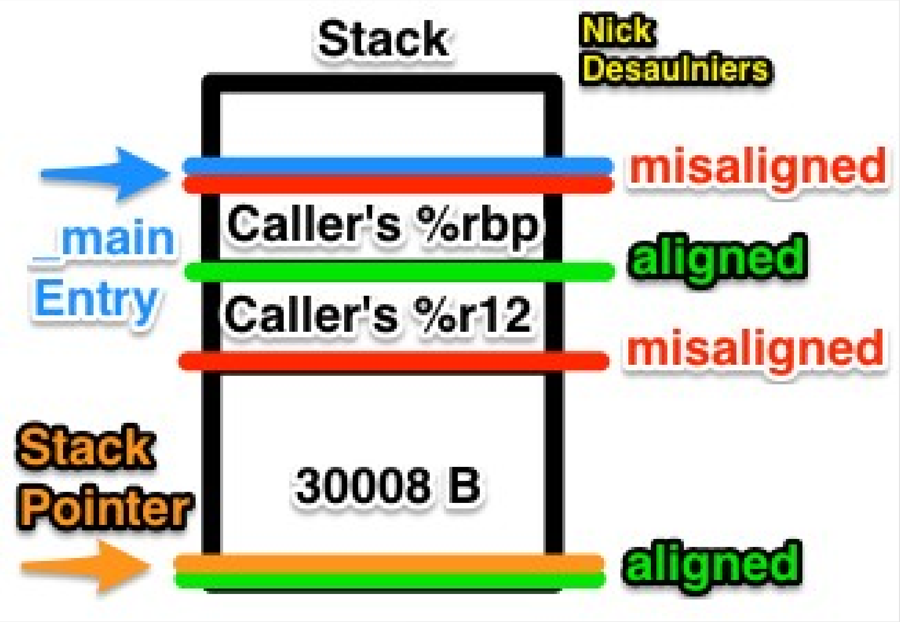
Epilogue for compiled x86_64

```
const char *const epilogue =
    "    addq $30008, %rsp\n" // clean up tape from stack.
    "    popq %r12\n" // restore callee saved register
    "    popq %rbp\n"
    "    ret\n";
```

- During the linking phase, we ensure linking in libc so we can call memset. What we do is backing up callee saved registers we’ll be using, stack allocating the tape, realigning the stack, copying the address of the only item on the stack into a register for our first argument, setting the second argument to the constant 0, the third arg to 30000, then calling memset.
- Finally, we use the callee saved register %r12 as our instruction pointer, which is the address into a value on the stack.

Alignment

- We can expect the call to memset to result in a segfault if simply subtract just 30000B, and not realign for the 2 registers (64 b each, 8 B each) we pushed on the stack.
- The first pushed register aligns the stack on a 16 B boundary, the second misaligns it; that's why we allocate an additional 8 B on the stack.



Data Pointers are straightforward

- Moving the instruction pointer (**>**, **<**) and modifying the pointed to value (**+**, **-**)

```
case '>':
  puts("  inc %r12");
  break;
case '<':
  puts("  dec %r12");
  break;
```

```
case '+':
  puts("  incb (%r12)");
  break;
case '-':
  puts("  decb (%r12)");
  break;
```

Output

- Have to copy the pointed to byte into the register for the first argument to putchar.
- We explicitly zero out the register before calling putchar, since it takes an int (32 b), but we're only copying a char (8 b) (C's type promotion rules).
  - x86-64 has an instruction that does both, movzXX, Where the first X is the source size (b, w) and the second is the destination size (w, l, q).
  - **`movzbl`** moves a byte (8 b) into a double word (32 b). %rdi and %edi are the same register, but %rdi is the full 64 b register, while %edi is the lowest (or least significant) 32 b.

```
case '.':
  puts("    movzbl (%r12), %edi");
  puts("    call putchar");
  break;
```

Input

- Easily call getchar, move the resulting lowest byte into the cell pointed to by the instruction pointer.
  - %al is the lowest 8 b of the 64 b %rax register

```
case ',':
    puts("  call getchar");
    puts("  movb %al, (%r12)");
    break;
```

Loop constructs [

- Have to match up jumps to matching labels
  - labels must be unique.
- Instead, whenever we encounter an opening brace, push a monotonically increasing number that represents the numbers of opening brackets we've seen so far onto a stack like data structure.
  - compare and jump to what will be the label that should be produced by the matching close label.
  - insert our starting label, and finally increment the number of brackets seen.

```
case '[':
    stack_push(&stack, num_brackets);
    puts  ("  cmpb $0, (%r12)");
    printf("  je bracket_%d_end\n", num_brackets);
    printf("bracket_%d_start:\n", num_brackets++);
    break;
```

Loop constructs]

- pop the number of brackets seen (or rather, number of pending open brackets which we have yet to see a matching close bracket) off of the stack, do our comparison, jump to the matching start label, and finally place our end label.

```
case ']':
    stack_pop(&stack, &matching_bracket);
    puts("  cmpb $0, (%r12)");
    printf("  jne bracket_%d_start\n", matching_bracket);
    printf("bracket_%d_end:\n", matching_bracket);
    break;
```

Code generation of Nested Loop

[[ ]]

```
  cmpb $0, (%r12)
  je bracket_0_end
bracket_0_start:

  cmpb $0, (%r12)
  je bracket_1_end
bracket_1_start:

  cmpb $0, (%r12)
  jne bracket_1_start
bracket_1_end:

  cmpb $0, (%r12)
  jne bracket_0_start
bracket_0_end:
```

Verify x86_64 compiler

- build

```
$ make && ./compiler-x64 progs/hello.b > hello.s
$ gcc -o hello-x64 hello.s && ./hello
Hello World!
```

- Check interpreter.c which reads a byte and immediately performs an action based on the operator.
- Benchmark on GNU/Linux for x86_64

```
$ ./compiler-x64 progs/mandelbrot.b > mandelbrot.s
$ gcc -o mandelbrot mandelbrot.s && time ./mandelbrot
real    0m9.366s
```

Brainf*ck Compiler

[ARM backend]

Prologue/Epilogue for ARM

```
const char *const prologue =
    ".globl main\n"
    "main:\n"
    "LDR R4 ,= _array\n"
    "push {lr}\n";

const char *const epilogue =
    "    pop {pc}\n"
    ".data\n"
    ".align 4\n"
    "_char: .asciz \"%c\"\n"
    "_array: .space 30000\n";
```

Data Pointers

- Moving the instruction pointer (**>**, **<**) and modifying the pointed to value (**+**, **-**)

```
case '>':
  puts("ADD R4, R4, #1");
  break;
case '<':
  puts("SUB R4, R4, #1");
  break;
```

```
case '+':
  puts("LDRB R5, [R4]");
  puts("ADD R5, R5, #1");
  puts("STRB R5, [R4]");
  puts("STRB R5, [R4]");
  break;
case '-':
  puts("LDRB R5, [R4]");
  puts("SUB R5, R5, #1");
  puts("STRB R5, [R4]");
  break;
```

Input/Output

```
case '.':
  puts("BL getchar");
  puts("STRB R0, [R4]");
  break;
```

NOTE:in epilogue
`"_char: .asciz \"%c\"\n"`

```
case '.':
  puts("LDR R0 ,= _char ");
  puts("LDRB R1, [R4]");
  puts("BL printf");
  break;
```

Loop constructs [

```
case '[':
    stack_push(&stack, num_brackets);
    printf("_in_%d:\n", num_brackets);
    puts  ("LDRB R5, [R4]");
    puts  ("CMP R5, #0");
    printf("BEQ _out_%d\n", num_brackets);
    num_brackets++;
    break;
```

Loop constructs]

```
case ']':
    stack_pop(&stack, &matching_bracket);
    printf("_out_%d:\n", matching_bracket);
    puts  ("LDRB R5, [R4]");
    puts  ("CMP R5, #0");
    printf("BNE _in_%d\n", matching_bracket);
    break;
```

Brainf*ck JIT Compiler

[x86_64 backend]

Our libc functions

- pass the address of our libc functions (memset, putchar, and getchar) into our JIT'ed function at runtime. This avoids those kooky stub functions you might see in a disassembled executable. That means we'll be invoking our JIT'ed function like:

```
typedef void* fn_memset (void*, int, size_t);

typedef int fn_putchar (int);

typedef int fn_getchar ();

void (*jitted_func)(fn_memset, fn_putchar,

                    fn_getchar) = mem;

jitted_func(memset, putchar, getchar);
```