intro to WebAssembly

ggu

what is WebAssembly?

- not rly web
- not rly assembly

WebAssembly (abbreviated Wasm) is a binary instruction format for a stack-based virtual machine. Wasm is designed as a portable compilation target for programming languages, enabling deployment on the web for client and server applications.

what does this mean

- binary instruction format -- like a machine code
- stack-based -- not register-based
- virtual machine -- does not run directly on hardware
  - e.g. if...else blocks would not run well on hardware
- portable -- can be run anywhere
- compilation target -- not designed for direct programming

why should I care

- wasm is increasingly used on the web
- it's also being used to run serverless applications in the cloud
- it's also being used for blockchain apps
- it's the "SFI of the future" -- bovav

demo

git clone https://github.com/emscripten-core/emsdk.git

cd emsdk

./emsdk install latest

./emsdk activate latest

source ./emsdk_env.sh

em++ test.cc -s WASM=1 -o hello.html

python3 -m http.server

how to look at wasm

```
$ wasm2wat test.wasm
(module
    ...
)
```

stack machine semantics

- similar to postfix notation
  - (5 - (1 + 2)) ==> 5 1 2 + -
  - no need for parentheses
- stack can only contain i32's and i64's

```
i32.const 5

i32.const 1

i32.const 2

i32.add

i32.sub
```

S-expressions

- looks like lisp (needs more parentheses)
- take the postfix & turn it into infix

```
(i32.sub (i32.const 5) (i32.add (i32.const 1) (i32.const 2)))
```

globals

```
$ wasm2wat test.wasm
(module
   (global (mut i32) (i32.const 68400))
   ...
   get_global 0 (pushes i32 68400 onto stack)
   ...
)
```

functions

```
$ wasm2wat test.wasm
(module
   (func $a (param i32 i32) (result i32)
      (i32.const 0) ; return value is what's left on stack
   )
)
```

calling a function

```
(func $a (param i32 i32) (result i32)

...

i32.const 2

i32.const 4

call $a

; 0 is now on the stack
```

locals

```
(func $add (param i32 i32 i32) (result i32)
   (local i32)
   get_local 0 ; first param
   get_local 1 ; second param
   get_local 2 ; third param
   i32.add
   i32.add
   tee_local 3 ; first local
)
```

loads/stores

- code is not part of memory
- each module gets its own memory space, initialized to zeros
  - indexed using i32's
- i32.load8_s
  - pops an i32 off the stack (the address)
  - loads 8 bit value at this address
  - sign-extends to 32 bits
  - pushes result to stack

C-stack

- wasm stack can only hold i32's and i64's
- what about arrays/structs/...?
- C-stack lives at the top of linear memory
  - first global is the "stack pointer"
- to allocate an array:
  - subtract size from c-stack ptr
  - array ptr is now the c-stack ptr's current position
- not actually part of wasm

if statements

```
(func $ternary (param i32 i32 i32) (result i32)
   (local i32)
   get_local 0
   if
      get_local 1
      set_local 3
   else
      get_local 2
      set_local 3
   end
   get_local 3 ;; place local 3 back onto the stack as the result
)
```

block & loop

```
(func $fact (param i32 $n) (result i32)
   (local (i32 $i) (i32 $res))
   i32.const 0
   tee_local $i ;; i = 0
   set_local $res ;; res = 0
   block
      loop
         (i32.gt_u (get_local $i) (get_local $n))
         br_if 1 ;; if i > n break
         (i32.mul (get_local $res) (get_local $i))
         set_local 2 ;; res = res * i
         (i32.add (get_local $i) (i32.const 1))
         set_local 1 ;; i = i + 1
         br 0
      end
   end
   get_local $res ;; return res
)
```

wasm compilation demo

```
clang --target=wasm32 -O2 --no-standard-libraries
-Wl,--export-all -Wl,--no-entry fib.c -o fib.wasm

(async() => {
    const response = await fetch('fib.wasm');
    const bytes = await response.arrayBuffer();
    const { instance } = await WebAssembly.instantiate(bytes);

    console.log('The answer is: ' + instance.exports.fib(5));
})();
```

compiling to WASI

WASI allows Wasm to work outside the browser!

Allows Wasm to make syscalls through the runtime

https://depth-first.com/articles/2019/10/16/compiling-c-to-webassembly-and-running-it-without-emscripten/

security properties

- control-flow integrity
  - most calls are well-typed & statically verified
  - all branching is well-structured
- stack-corruption resistant
  - wasm stack does not contain any buffers
  - no rop :-(
  - C-stack is not stack-corruption resistant, can overflow stack vars
- not heap-corruption resistant!
  - data-based heap corruption should work
  - but function pointers go through a layer of indirection (function tables)
  - so you can only replace a vptr with another valid vptr

useful wasm tools

- wasmtime
  - runs WASI binaries
- wasm-decompile
  - not always super helpful
- wasm2c
- wasm2wat
- ghidra-wasm-plugin