WEBASSEMBLY illustrated

exploring some mental models and implementations

Takenobu T.
NOTE

- Please refer to the official documents in detail.
- This information is based on "WebAssembly Specification Release 1.0 (Draft, last updated Oct 31, 2018)".
- This information is current as of Nov, 2018. Still work in progress.
1. Introduction
1. Introduction

Overview
WebAssembly is a code format.

WebAssembly is a safe, portable, low-level code format.

References: [1] Ch.1.1, [2], [3], [6]
WebAssembly encodes a low-level, assembly-like programming language.
WebAssembly has multiple concrete representations. (its text format and the binary format.)

References: [1] Ch.2, Ch.5, Ch.6, [7]
WebAssembly is a virtual instruction set architecture (virtual ISA). Execution behavior is defined in terms of an abstract machine.

References: [1] Ch.1.1, [2], [3]
Validation checks that a WebAssembly module is well-formed. Validity is defined by a type system. The type system of WebAssembly is sound, implying both type safety and memory safety with respect to the WebAssembly semantics.

References: [1] Ch.1, Ch.3, Ch.7, [2]
2. WebAssembly abstract machine
2. WebAssembly abstract machine

Abstract machine
WebAssembly abstract machine is based on a stack machine. The abstract machine includes a store and an implicit stack.

References: [1] Ch.2, Ch.4, [2], [4]
2. WebAssembly abstract machine

Store
The store represents all global state. The store have been allocated during the life time of the abstract machine.

References: [1] Ch.2, Ch.4, [2], [4]
The function component of a module defines a vector of functions. Functions are referenced through function indices.
The table is an array of opaque values of a particular element type. Currently, the only available element type is an untyped function reference. This allows emulating function pointers by way of table indices. Tables are referenced through table indices.
The linear memory is a contiguous, mutable array of raw bytes. The linear memory can be addressed at byte level (including unaligned). The size of the memory is a multiple of the WebAssembly page size.

References: [1] Ch.2, Ch.4, [2], [4]
The globals component defines a vector of global variables. The globals are referenced through global indices. The global variables hold a value and can either be mutable or immutable.
2. WebAssembly abstract machine

Stack
Most instructions interact with the implicit stack. The stack contains values, labels and frames (activations).

References: [1] Ch.2, Ch.4, [2], [4]
Instructions manipulate values on an implicit operand stack. The layout of the operand stack can be statically determined at any point in the code.
Each structured control instruction introduces an implicit label. Labels are targets for branch instructions that reference them with label indices.

References: [1] Ch.2, Ch.4, [2], [4]
Frames hold the values of its local variables (including arguments). Frames also carry the return arity of the respective function.
2. WebAssembly abstract machine

Computational model
Computational model

WebAssembly abstract machine

Instruction sequence → Instruction
operation → Instruction

Function sequence

Operand stack
Control stack
Call stack (Frame)
Linear memory
Globals
Function
Table
Store

References: [1] Ch.2, Ch.4, [2], [4]
Computational model

WebAssembly abstract machine

Instruction sequence

operation

add, sub, ...
block, br, ...
call, return

instruction

Operand stack
values

Control stack
labels

Functions
code

Linear memory
load, store

Globals
global vars

Call stack (Frame)
local vars

Store

Table

References: [1] Ch.2, Ch.4, [2], [4]
2. WebAssembly abstract machine

Type
WebAssembly provides only four basic value types. 32 bit integers also serve as Booleans and as memory addresses.
Instructions have type annotations.

Some instructions have type annotations. For example, the instruction `i32.add` has type `[i32 i32] → [i32]`, consuming two `i32` values and producing one.

References: [1] Ch.2, Ch.3, Ch.4, [2]
Functions have type declarations

Each function takes a sequence of WebAssembly values as parameters and returns a sequence of values as results as defined by its function type.
Control blocks have also a type declaration

Every control construct is annotated with a function type.

References: [1] Ch.2, Ch.3, Ch.4, [2]
2. WebAssembly abstract machine

Trap
Certain instructions may produce a trap, which immediately aborts execution. Traps cannot be handled by WebAssembly code, but are reported to the outside environment, where they typically can be caught.

References: [1] Ch.1, Ch.4, [2], [4]
WebAssembly abstract machine

Instruction sequence

function

operation

unreachable
zero division
invalid conversion

out of bounds

undefined element
uninitialized element

Trap

References: [1] Ch.1, Ch.4, [2], [4]
A trap occurs if an access is not within the bounds of the current memory size.
2. WebAssembly abstract machine

Thread
The current version of WebAssembly is single-threaded, but configurations with multiple threads may be supported in the future.

References: [1] Ch.4, [2], [4]
2. WebAssembly abstract machine

External interface
Functions, table, memory and globals may be shared via import/export.

References: [1] Ch.2, Ch.4. [2]. [5]
Table, memory and globals can be mutated by external mean.

References: [1] Ch.2, Ch.4. [2]
Any exported function can be invoked externally.

References: [1] Ch.2, Ch.4. [2]
Foreign call

Call instructions can invoke an imported function.

References: [1] Ch.2, Ch.4. [2]
3. WebAssembly module
3. WebAssembly module

Module
WebAssembly programs are organized into modules. Modules are the distributable, loadable, and executable unit of code. WebAssembly modules are distributed in a binary format.

References: [1] Ch.1, Ch.2, Ch.4, [2], [4], [5]
A module collects definitions for types, functions, table, memory, and globals. In addition, it can declare imports and exports and provide initialization logic in the form of data and element segments or a start function.

References : [1] Ch.1, Ch.2, Ch.4, [2], [4], [5]
A module corresponds to the static representation of a program. A module instance corresponds to a dynamic representation.

References: [1] Ch.1, Ch.2, Ch.4, [2], [4], [5]
3. WebAssembly module

Binary encoding
Binary encoding of modules

WebAssembly module (WebAssembly binary)

```
00 61 73 6d 01 00 00 00 01 07 01 60 02 7e 7e 01 7e 03 ...
```

Form

sections

- magic
- version
- import section
- type section
- table section
- func section
- :
- code section

The binary encoding of modules is organized into sections.

References: [1] Ch.5, [7], [5]
Sections

Binary encoding of modules

Each section consists of
- a one-byte section id,
- the u32 size of the contents, in bytes,
- the actual contents, whose structure is depended on the section id.

References : [1] Ch.5, [7], [5]
Example of WebAssembly module

[Text format]
(module
  (func (export "foo" (result i32)
    i32.const 7)))

[Binary format]

0000000: 0061 736d        ; WASM_BINARY_MAGIC
0000004: 0100 0000        ; WASM_BINARY_VERSION
; section "Type" (1)
0000008: 01               ; section code
0000009: 05               ; section size
000000a: 01               ; num types
; type 0
000000b: 60               ; func
000000c: 00               ; num params
000000d: 01               ; num results
000000e: 7f               ; i32
; section "Function" (3)
000000f: 03               ; section code
0000010: 02               ; section size
0000011: 01               ; num functions
0000012: 00               ; function 0 signature
; index

; section "Export" (7)
0000013: 07               ; section code
0000014: 07               ; section size
0000015: 01               ; num exports
0000016: 03               ; string length
0000017: 666f 6f           ; foo ; export name
000001a: 00               ; export kind
000001b: 00               ; export func index
; section "Code" (10)
000001c: 0a               ; section code
000001d: 06               ; section size
000001e: 01               ; num functions
; function body 0
000001f: 04               ; func body size
0000020: 00               ; local decl count
0000021: 41               ; i32.const
0000022: 07               ; i32 literal
0000023: 0b               ; end

References: [1] Ch.5, Ch.6, [7], [5]
All integers are encoded using the LEB128 variable-length integer encoding.
4. WebAssembly instructions
4. WebAssembly instructions

Instructions
Instructions fall into two main categories. Simple instructions perform basic operations on data. Control instructions alter control flow.
4. WebAssembly instructions

Simple instructions
Numeric instructions pop arguments from the operand stack and push results back to it.
The const instruction pushes the value to the stack.
The drop instruction simply throws away a single operand.
The select instruction selects one of its first two operands based on whether its third operand is zero or not.
Global variable instructions get or set the values of variables.

References: [1] Ch.2, Ch.4, [2], [4]
Local variable instructions get or set the values of variables. (including function arguments)
Memory is accessed with load and store instructions for the different value types.

References: [1] Ch.2, Ch.4, [2], [4]
The memory.grow instruction grows memory by a given delta. The memory.grow instruction operates in units of page size (64KiB).
4. WebAssembly instructions

Control instructions
Control flow is expressed with well-nested constructs such as blocks, loops, and conditionals (if-else).
Structured control flow allows simpler and more efficient verification.

References: [1] Ch.2, Ch.4, [2], [4]
Structured control instructions

The block, loop and if instructions are structured control instructions.

References: [1] Ch.2, Ch.4, [2], [4]
Branches can only target control constructs. Intuitively, a branch targeting a block or if behaves like a break statement, while a branch targeting a loop behaves like a continue statement.
Branches have "label" immediates.
It do not reference program positions in the instruction stream
but instead reference outer control constructs by relative nesting depth.
The br_if instruction performs a conditional branch.
The br_table performs an indirect branch through an operand indexing into the label vector.
The call instruction invokes another function, consuming the necessary arguments from the stack and returning the result values of the call.
The call_indirect instruction calls a function indirectly through an operand indexing into a table.
The return instruction is an unconditional branch to the outermost block, which implicitly is the body of the current function.

References: [1] Ch.2, Ch.4, [2], [4]
4. WebAssembly instructions

Byte order
WebAssembly abstract machine is little endian byte order. When a number is stored into memory, it is converted into a sequence of bytes in little endian byte order.
Appendix A

Semantics
Validation and execution semantics

The semantics is derived from the following article: "Bringing the Web up to Speed with WebAssembly" [2]

Validation semantics: typing rules

Figure 3. Typing rules

Figure 2. Small-step reduction rules

References: [2], [1] Ch.3, Ch.4, Ch.7
Appendix B

Implementations
let rec step (c : config) : config =
  let {frame; code = vs, es; _} = c in
  let e = List.hd es in
  let vs', es' =
    match e.it, vs with
    | Plain e', vs ->
      (match e', vs with
       | Unreachable, vs ->
         vs, [Trapping "unreachable executed" @@ e.at]
       | Nop, vs ->
         vs, []
       | Block (ts, es'), vs ->
         vs, [Label (List.length ts, [], ([], List.map plain es')) @@ e.at]
       | Loop (ts, es'), vs ->
         vs, [Label (0, [e' @@ e.at], ([], List.map plain es')) @@ e.at]
       | If (ts, es1, es2), I32 0l : : vs' ->
         vs', [Plain (Block (ts, es2)) @@ e.at]
Interpreter : WABT

https://github.com/WebAssembly/wabt
[src/interp/interp.cc]

```cpp
Result Thread::Run(int num_instructions) {
  Result result = Result::Ok;

  const uint8_t* istream = GetIstream();
  const uint8_t* pc = &istream[pc_];
  for (int i = 0; i < num_instructions; ++i) {
    Opcode opcode = ReadOpcode(&pc);
    assert(!opcode.IsInvalid());
    switch (opcode) {
      case Opcode::Select: {
        uint32_t cond = Pop<uint32_t>();
        Value false_ = Pop();
        Value true_ = Pop();
        CHECK_TRAP(Push(cond ? true_ : false_));
        break;
      }
      case Opcode::Br:
        GOTO(ReadU32(&pc));
        break;
      case Opcode::BrIf:
    }
```
// Decode the WebAssembly opcodes and emit LLVM IR for them.
OperatorDecoderStream decoder(functionDef.code);
UnreachableOpVisitor unreachableOpVisitor(*this);
OperatorPrinter operatorPrinter(irModule, functionDef);
Uptr opIndex = 0;
while(decoder && controlStack.size())
{
    irBuilder.SetCurrentDebugLocation(
        llvm::DILocation::get(llvmContext, (unsigned int)opIndex++,
        0, diFunction));
    if(ENABLE_LOGGING)
    { logOperator(decoder.decodeOpWithoutConsume(operatorPrinter)); } 

    if(controlStack.back().isReachable) { decoder.decodeOp(*this); } 
    else 
    { 
        decoder.decodeOp(unreachableOpVisitor);
    }
    wavmAssert(irBuilder.GetInsertBlock() == returnBlock);
}

if(EMIT_ENTER_EXIT_HOOKS)
switch (op.b0) {
    case uint16_t(Op::End):
        if (!emitEnd()) {
            return false;
        }

        if (iter_.controlStackEmpty()) {
            if (!deadCode_) {
                doReturn(funcType().ret(), PopStack(false));
            }
            return iter_.readFunctionEnd(iter_.end());
        }
    NEXT();

    // Control opcodes
    case uint16_t(Op::Nop):
        CHECK_NEXT(iter_.readNop());
    case uint16_t(Op::Drop):
        CHECK_NEXT(emitDrop());
    case uint16_t(Op::Block):
        CHECK_NEXT(emitBlock());
    case uint16_t(Op::Loop):
Assemble Wasm text format (.wat) to Wasm binary format (.wasm):

**Binaryen:**

```
$ wasm-as sample.wat
```

**WABT:**

```
$ wat2wasm sample.wat
```

```
$ wat2wasm -v sample.wat
```

**Spec:**

```
$ wasm -d sample.wat
```
Disassemble Wasm binary format (.wasm) to Wasm text format (.wat)

Binaryen:

$ wasm-dis sample.wasm

WABT:

$ wasm2wat sample.wasm

$ wasm-objdump -d sample.wasm

Spec:

$ wasm -d sample.wasm
Desugar

Desugar Wasm text format (.wat) to Wasm text format (.wat)

WABT:

```
$ wat-desugar sample.wat
```
Dump information

Dump Wasm binary format (.wasm) information:

WABT:

```
$ wasm-objdump -s sample.wasm
```

```
$ wasm-objdump -x sample.wasm
```

Spec:

```
$ wasm -s sample.wasm
```
Run Wasm binary format (.wasm) and Wasm text format (.wat):

**WABT : Run Wasm binary format with trace**

```
$ wasm-interp --run-all-exports --trace sample.wasm
```

**WAVM : Run Wasm text format**

```
$ wavm-run sample.wat
```

**Spec : Run Wasm binary format**

```
$ wasm sample.wasm -e '(invoke "XXX")'
```
REPL (Read-Eval-Print-Loop):

Spec:

```
$ wasm -
```

```
$ wasm sample.wasm -
```
Appendix B

Test suite
# Test suite and Wasm text format examples

[https://github.com/WebAssembly/spec][test/core]

<table>
<thead>
<tr>
<th>README.md</th>
<th>fac.wast</th>
<th>names.wast</th>
</tr>
</thead>
<tbody>
<tr>
<td>address.wast</td>
<td>float_exprs.wast</td>
<td>nop.wast</td>
</tr>
<tr>
<td>align.wast</td>
<td>float_literals.wast</td>
<td>return.wast</td>
</tr>
<tr>
<td>binary.wast</td>
<td>float_memory.wast</td>
<td>run.py*</td>
</tr>
<tr>
<td>block.wast</td>
<td>float_misc.wast</td>
<td>select.wast</td>
</tr>
<tr>
<td>br.wast</td>
<td>forward.wast</td>
<td>set_local.wast</td>
</tr>
<tr>
<td>br_if.wast</td>
<td>func.wast</td>
<td>skip-stack-guard-page.wast</td>
</tr>
<tr>
<td>br_table.wast</td>
<td>func_ptrs.wast</td>
<td>stack.wast</td>
</tr>
<tr>
<td>break-drop.wast</td>
<td>get_local.wast</td>
<td>start.wast</td>
</tr>
<tr>
<td>call.wast</td>
<td>globals.wast</td>
<td>store_retval.wast</td>
</tr>
<tr>
<td>call_indirect.wast</td>
<td>i32.wast</td>
<td>switch.wast</td>
</tr>
<tr>
<td>comments.wast</td>
<td>i64.wast</td>
<td>tee_local.wast</td>
</tr>
<tr>
<td>const.wast</td>
<td>if.wast</td>
<td>token.wast</td>
</tr>
<tr>
<td>conversions.wast</td>
<td>imports.wast</td>
<td>traps.wast</td>
</tr>
<tr>
<td>custom.wast</td>
<td>inline-module.wast</td>
<td>type.wast</td>
</tr>
<tr>
<td>data.wast</td>
<td>int_exprs.wast</td>
<td>typecheck.wast</td>
</tr>
<tr>
<td>elem.wast</td>
<td>int_literals.wast</td>
<td>unreachable.wast</td>
</tr>
<tr>
<td>endianness.wast</td>
<td>labels.wast</td>
<td>unreached-invalid.wast</td>
</tr>
<tr>
<td>exports.wast</td>
<td>left-to-right.wast</td>
<td>unwind.wast</td>
</tr>
<tr>
<td>f32.wast</td>
<td>linking.wast</td>
<td>utf8-custom-section-id.wast</td>
</tr>
<tr>
<td>f32_bitwise.wast</td>
<td>loop.wast</td>
<td>utf8-import-field.wast</td>
</tr>
<tr>
<td>f32_cmp.wast</td>
<td>memory.wast</td>
<td>utf8-import-module.wast</td>
</tr>
<tr>
<td>f64.wast</td>
<td>memory_grow.wast</td>
<td>utf8-invalid-encoding.wast</td>
</tr>
<tr>
<td>f64_bitwise.wast</td>
<td>memory_redundancy.wast</td>
<td></td>
</tr>
<tr>
<td>f64_cmp.wast</td>
<td>memory_trap.wast</td>
<td></td>
</tr>
</tbody>
</table>

---

Note: `.wast` extension means command-script and Wasm text format.
Appendix B

Desugar examples
Desugar example

Text format
syntactic sugar
(func (result i32)
  (i32.add
   (i32.const 1)
   (i32.const 2)))

Text format
core syntax
(func (result i32)
  i32.const 1
  i32.const 2
  i32.add)
Desugar example

Text format

syntactic sugar

(func (result i32)
  (i32.add
   (i32.const 1)
   (i32.mul
    (i32.const 2)
    (i32.const 3))))

Text format

core syntax

(func (result i32)
  i32.const 1
  i32.const 2
  i32.const 3
  i32.mul
  i32.add)

References: [7], [1] Ch.6, Ch.2, Ch.4
Desugar example

Text format
syntactic sugar

(func (result i32)
 (block (result i32)
  (i32.add
   (i32.const 1)
   (i32.const 2)))))

Text format
core syntax

(func (result i32)
 block (result i32)
 i32.const 1
 i32.const 2
 i32.add
 end)
Desugar example

Text format
syntactic sugar
(func (block $label_a (block $label_b br $label_a)))

Text format
core syntax
(func block block br 1 end end)
Desugar example

Text format
syntactic sugar
(func (result i32)
  (if (result i32) (get_global 0)
    (then (i32.const 1))
    (else (i32.const 2)))
)

Text format
core syntax
(func (result i32)
  get_global 0
  if (result i32)
    i32.const 1
  else
    i32.const 2
  end)

References: [7], [1] Ch.6, Ch.2, Ch.4
Appendix C

Future
Future directions

* zero-cost exception, threads, SIMD

* tail call, stack switching, coroutines

* garbage collectors

References: [2], [3], [4]
References
References

https://webassembly.github.io/spec/core/

[2] Bringing the Web up to Speed with WebAssembly

[3] WebAssembly High-Level Goals
https://webassembly.org/docs/high-level-goals/

https://webassembly.org/docs/rationale/

https://webassembly.org/docs/modules/


[7] MDN: Understanding WebAssembly text format
https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

https://en.wikipedia.org/wiki/LEB128
References

https://github.com/WebAssembly/spec

[C2] Binaryen: Compiler infrastructure and toolchain library for WebAssembly, in C++
https://github.com/WebAssembly/binaryen

[C3] WABT: The WebAssembly Binary Toolkit
https://github.com/WebAssembly/wabt

[C4] WAVM: WebAssembly Virtual Machine
https://github.com/WAVM/WAVM

[C5] mozilla/gecko-dev (Firefox)
https://github.com/mozilla/gecko-dev
Here is the slide: https://github.com/takenobu-hs/WebAssembly-illustrated