WEBASSEMBLY

Travis Banken
Computer Science
University of Arizona
Tucson, AZ 85721
bankent1@email.arizona.edu

Ben Schroeder
Computer Science
University of Arizona
Tucson, AZ 85721
btschroeder@email.arizona.edu

April 19, 2020

ABSTRACT

WebAssembly is neither assembly, nor constrained to only the web.

1 Introduction

2 History

The goal of WebAssembly (Wasm) was to be a portable compilation target for higher level languages. Some examples of these languages are C, C++, and Rust. Essentially, the browser is already very portable across platforms, so Wasm aims to provide a way to port programs written in less portable languages (like C) to the "Browser Platform". Wasm was a joint effort from WebAssembly CG members from the four major browsers: Chrome, Edge, WebKit, and Firefox. It was made standard by the W3C.

The WebAssembly Community Group was started in April of 2015. Wasm was made ready to ship to all browsers in March of 2017. Wasm is supported on the four major browsers and is ready for use.

3 Control Structures

3.1 if-else

The if-else structure in WebAssembly is very similar to other languages. The if statement takes in a condition and if the condition is true, it will execute the body of code given. Below we can see an example of this in action.

```
(if
    (i32.eq (local.get $x) (i32.const 0))
    (then
        CODE FOR TRUE CASE
    )
    (else
        CODE FOR FALSE CASE
    )
)
```

As the WebAssembly text format is written using S-expressions, the format looks closer to function programing languages. We see that the first argument to the if function is a condition. This condition is provided by the .eq method for i32 types and checks if a local var is equal to 0.

3.2 br, br_if, br_table

The br, br_if, and br_table structures are used to branch to various levels of nested blocks/loops. These levels are given by indices, where 0 is the index of the level currently in, and each index higher is an outer-block. Branching to the highest level returns from the function. br is an unconditional branch (branches to given level no matter what).

```
(block ... (br 0) ... )
```

Here, the second ... will not get executed, since branching to a block goes to the end of a block.

br_if is a conditional branch and accepts an i32 type along with block level index. If the given value is non-zero it branches, otherwise ignored.

```
(block ... (br_if 0 CONDITION) ...)
```

In this case, the second ... will get executed only if CONDITION is false, otherwise the branch will go to the end of the block.

Finally the br_table instruction acts a bit like a switch statement in C. It is given a list of integers referring to various block depths to branch to, and accepts another integer which is an index for which block depth in the list to branch to. This will make more sense with an example.

```
(block ...
    (block ...
        (block ...
            (BRANCH INDEX)
            (br_table 2 1 0 3)
            ...
```

If BRANCH INDEX pushed 0, 1, 2 onto the stack, br_table will branch to the outer-most block, second outer-most, and inner-most block respectively. If some other value was pushed on the stack, br_table default branches to depth 3 which is whatever structure is above the first block statement (could be another block, or the function, in which case would return).

3.3 loop

The loop structure allows for repeated iteration over some instructions. The above branching instructions are used with loop as well as with block, the only difference being, when branching to a loop, it branches to the start of the loop, versus the end of a block. Here's an example:

```
(loop ... (br 0))
```

This would execute the instructions in ... repeatedly forever...

```
(loop ... (br_if 1 CONDITION) ... (br 0))
```

This behaves like a while loop which breaks when CONDITION is true.

3.4 other

There are a few small control structure tools that work with the big forms mentioned above. First there is the nop instruction. This is exactly what the name implies, it does nothing. Next, there is the block. This allows for organization of a sequence of instructions with a label at the end of the block. Additionally, there is an end instruction. This marks the end of a block, loop, or function. Finally, there a return instruction which allows for early exits in functions.

4 Data Types

WebAssembly only has 4 data types: i32, i64, f32, f64. This might seem limiting at first, but like assemby languages, these types can represent both values and addresses. This allows the programmer/compiler to build data structures as complex as they need to be, so it is more powerful than what it seems.

4.1 i32, i64

i32 can hold 32-bit signed integers while i64 can hold 64-bit signed integers. However, the operation on the value is what actually determines if the number is signed or not. In essence, these types are just numbers, with the complexity coming from the operations which can be performed on them. Here are some examples on i-operations:

```
i32.const 35
i32.const 28
i32.add
```

4.2 f32, f64

f32 can hold 32-bit floating point values while f64 can hold 64-bit floating point values. Another way of looking at it is f32 are single precision floating point numbers and f64 are double precision floating point numbers. Here are some examples on f-operations:

```
(f32.add (f32.const 4.2) (f32.const 6.9))
(f32.sub (f32.const 1.2345) (f32.const 0.2345))
(f32.div (f32.const 99.9) (f32.const 3.0))
```

5 Subprograms

WebAssembly has basic means of creating subprograms. There are two layers: modules and functions. Modules are the base of all wasm files. Every function a programmer writes must be contained within a module. These modules are then loaded into javascript objects. So, you can not have a function without a module but you can have a module without any functions (though, this has only limited use). There is a sense of private and public functions. All functions can be considered private until they are explicitly exported using the export keyword. Once this is done, they can be called from javascript or other wasm modules. To use external functions in your module, you must import them. You can see in the program below, it needs to import a simple print function from javascript and then exports its own function back to javascript. All of this contained in a wasm module.

```
(module
    (import "terminal" "sprint" (func $sprint (param i32) (param i32))) ;;can import JS

    (import "terminal" "mem" (memory 1))
    (data (i32.const 0) "Each module contains functions which are exported to be called by
    (data (i32.const 76) "\n\00")

    (func $line ;;local functions can be called by other functions in the module
        (call $sprint (i32.const 76) (i32.const 2))
    )

    (func (export "p4_func") ;;export functions to be called by JS
        (call $sprint (i32.const 0) (i32.const 75))
        call $line
    )
)
```

6 Summary

References

[1] WebAssembly Community Group and Andreas Rossberg. *WebAssembly Specification*. Release 1.0. 25 March 2020. Retrieved from https://webassembly.github.io/spec/core/_download/WebAssembly.pdf

[2] Mozilla Developer Network. WebAssembly. Last modified 8 Feb 2020. Retrieved from https://developer.mozilla.org/en-US/docs/WebAssembly

[3] WebAssembly WebAssembly Developer's Guide Retrieved from https://webassembly.org/getting-started/developers-guide/