Web Assembly Virtual Machine

Computer Language Processing '17 Final Report

Deniz Ulusel Malte Tobias Brodmann Sabit Gokberk Karaca

EPFL

{deniz.ulusel, malte.brodmann, gokberk.karaca}@epfl.ch

1. Introduction

The first part of our compiler project, which we implemented during the semester, is composed of the following stages:

Lexer
This stage separates the high level code into tokens.

Parser
This stage processes tokens into a tree representation using a grammar. It also makes sure that input program satisfies the language grammar.

Name analyzer
This stage makes sure the variable names adhere to language rules and generates distinct symbols for each variable declaration.

Type checker
This stage checks that the input code adheres to typing rules of the language.

Code generation
This stage generates Web Assembly code from the the abstract syntax tree representation of a program passed to it by previous phases.

Interpreter
This stage interprets the abstract syntax tree representation of a program passed to it by previous phases.

Our project extends this compiler pipeline to enable the execution of Web Assembly modules on a virtual machine. We achieve this by adding another stage to the pipeline, called Virtual Machine. Virtual Machine takes as input `Module` objects generated by Code Generation stage. Unlike the Interpreter stage which executes programs using an AST representation, this new stage executes a Web Assembly representation of Amy programs. This is made possible by emulating a process virtual machine that is modeled after the stack machine specified by Web Assembly, albeit with reduced functionality. We attempted to implement this machine with low level, hardware ideas in mind: We utilize arrays as stacks and data memory with minimal abstraction. Despite the difference in approach, the output of executing an input program on our virtual machine should be identical to that produced by the Interpreter stage.

2. Examples

Using the extended compiler will result in the compilation of the input program and the execution of the generated WebAssembly code. Potential inputs and outputs will be displayed on the console.

As an example, a simple Amy program and its execution on the console is given.

```
object HelloInt extends App {
  Std.printString("What is your name?");
  val name: String = Std.readString();
  Std.printString("Hello " ++ name ++ "!");
  Std.printString("And how old are you?");
  val age: Int = Std.readInt();
  Std.printString(Std.intToString(age) ++ " years old then.")
}
```

```
sbt:amyc> run lib/Std.scala exmpls/test.scala
[info] Running amyc.Main lib/Std.scala exmpls/test.scala
-> What is your name?
<- James
-> Hello James!
-> And how old are you?
<- 25
-> 25 years old then.
```

3. Implementation

3.1 Theoretical Background

Brief descriptions of the concepts of virtual machine and WebAssembly are given in this section.

A virtual machine is a program that is based on a real computer architecture and emulates a physical computer in software. A process virtual machine is a virtual machine that is designed to execute the same representation of a program across different platforms [Wikipedia contributors 2017].

WebAssembly is a specification for an assembly langauge and a corresponding machine language that is meant to be run on a virtual machine in web browsers. With the goal of enabling languages such as C and C++ to be run on web browsers, as well as speeding up execution of web applications [Wikipedia contributors 2018].

3.2 Assumptions

The virtual machine is designed only to execute WebAssembly modules outputted by the Code Generation stage, not WebAssembly code in general. For this reason, the instructions it is able to correctly execute are limited to those that may possibly be outputted by the Code Generation stage. It also makes the following assumptions, which are guaranteed to be true for programs outputted by the Code Generation stage:

(1) All locals, globals and items on the main stack are of type `i32`.

(2) Typed operations are always of type `i32`.

(3) No `Loop` block is contained within either branch of an `If Else` block.

If for some input program one or more of these assumptions do not hold, the behavior of the virtual machine is undefined.

3.3 Implementation Details

3.3.1 VM Architecture

The virtual machine relies on five main data structures to execute given programs. The structures are defined as follows:

Instruction memory

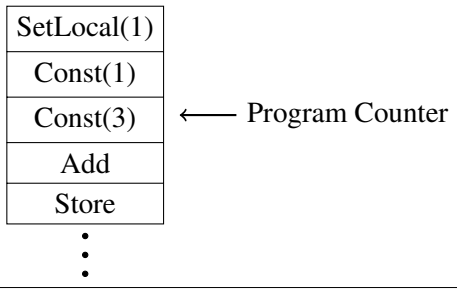

Figure 1. Instruction memory

The instruction memory is the memory that contains the program itself. It is modeled as an array of `Instruction` objects as defined in the `wasm` package. Its size is determined by the length of the input program after preprocessing. A pointer to the instruction memory, called the program counter, is used to keep track of the next instruction to be executed.

Data memory

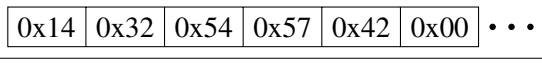

Figure 2. Data memory

The data memory is used to store and operate on strings and case class objects. It is modeled as an array of bytes with a fixed size. It can be addressed by bytes of four byte words. The words are stored in little endian order.

Main stack

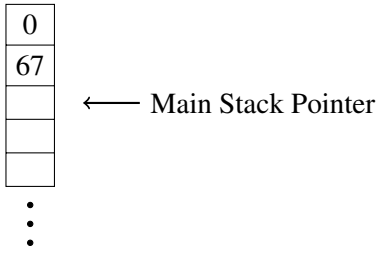

Figure 3. Main stack

The main stack is the working stack that is used by WebAssembly instructions. It is modeled as an array of integers, the size of which is fixed. A pointer to the main stack is used to keep track of the top of the stack, i.e. the first unused slot of the stack.

Call stack

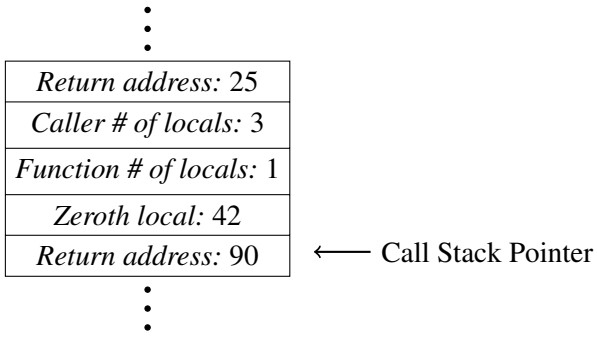

Figure 4. Call stack

The call stack is used to store data about functions that have been called but have not returned yet. It is modeled as an integer array with a fixed size. The data it contains is grouped into frames. A frame is a group of slots in the call stack with a minimum size of 3. The zeroth of these slots holds the return address for the function. The first slot holds the number of locals for the caller of this function, this data is used to correctly return from the function. The second slot holds the number of locals for the function itself. The rest of the slots, if there are any, hold the locals belonging to the function. A pointer to the call stack always points to the zeroth slot (the return address) of the frame which belongs to the currently executing function.

Global memory

98	2	71	333	1024	31	9999	0

Figure 5. Global memory

The global memory is used to store global variables. It is modeled as a fixed size integer array.

In addition to these main data structures, the virtual machine also utilizes some maps to keep track of the instruction memory indexes for functions, `Loop` labels and `If Else` blocks.

3.3.2 Code Execution

Before execution, the input code is preprocessed by first reordering functions to place the main function in the beginning. Then, in order to indicate its end, a `Return` instruction is inserted to the end of each function.

After preprocessing is done, all instructions of the input program are loaded into the instruction memory. This is done by placing all instructions in each function back to back in the instruction memory. Also, the program counter is initialized to zero.

The machine starts by executing the first instruction of the main method and continues until the last instruction of the main method is executed. After the execution of each instruction, an integer value is returned, which represents the index of the next instruction to be executed, i.e. new value of the program counter. Most instructions increment the program counter by one, however, instructions as `Call` or `Branch` may change the flow of the program by looking up their target index from the index maps and returning it. The program counter is updated to the integer value returned which results in the next instruction in the program being fetched and executed.

Although each of the instructions is handled differently by the virtual machine, we will separate them into groups based on their purposes to facilitate their discussion.

Numeric/Logical Instructions

All of these instructions consume two operands from the main stack and apply an operation on these operands. The top value of the main stack is used as the second operand of the instruction. The result is pushed onto the main stack after execution. Subsequently, the program counter is incremented by one and returned to continue with the next instruction. These instructions do not change the order in which the program are executed.

Control Instructions

These instructions mainly serve to change the flow of the program.

`If` instructions are used for branch decision. They are the most complex instruction in this group in terms of their implementation. To be able to execute `If` instructions correctly, a helper method is used, which is responsible for iterating over the `If Then Else` construct and finding the corresponding `Else` and `End` instructions. This helps us determine the beginning and

ending indices of this execution block. Here, we also take nested `If Then Else` constructs into account. Our assumption that `Loop` blocks are never inside of one of the `If Then Else` branches is useful at this point. Since both `If Then Else` blocks and `Loop` blocks end with an `End` instruction, we would have problems without this assumption. Before executing one of the two branches, a conditional value is consumed from the top of the main stack. If this value is different from zero, all instructions between `If` and `Else` are executed. Otherwise, all instructions between `Else` and `End` are executed. An exceptional case is when the `Branch` instruction occurs inside of an `If Then Else` construct. Upon encountering such a case, the execution of the block is stopped and the program counter is set to the target address.

In order to optimize the process of finding `If`, `Else` and `End` instructions that correspond to each other, a map is used. If the indices have been already found during a previous execution, they are simply retrieved from the map. If they don't exist in the map, the helper function is called and indices are added to the map.

`Loop` and `Branch` are two additional control instructions. The execution of these two instructions is also handled by using a map from labels to instruction indices. Whenever a `Loop` instruction is encountered, its label is added to the map with its index. Then `Branch` instructions are executed by looking up the mentioned label and updating the program to the value retrieved from the map.

`Call` and `Return` instructions are used in order to start and end the execution of a function. Before a `Call` instruction is executed, the parameters of the callee have to be pushed to the main stack. Since this is guaranteed by the Code Generator, it does not require any further actions. Upon executing the `Call` instruction, a new frame in the call stack has to be prepared. This frame contains the return address of the callee, the number of local variables of the caller and number of local variables of the callee. After pushing this new frame to the stack, the local variables of the callee can also be consumed from the main stack and placed into the new frame. This completes the preparation for a new function call and the program counter is updated to the starting index of the callee function. Starting indices of each function are stored in a map together with their names. The execution of a function ends when the `Return` instruction is encountered. This instruction retrieves the return address from the functions call frame and sets the program counter accordingly. At this point the frame of the finished function can be removed from the call stack and the execution of the caller function can continue.

Some I/O functions are treated as special cases of function calls since the Code Generation stage does not generate WebAssembly code for them. These functions are handled by using Scala's I/O libraries and some utility functions.

`Std_printInt` is implemented by consuming one value from main stack and using Scala's `println` function to print the value on the console.

`Std_printString` is implemented by consuming one value from the stack. This value is the start address of the string value in the memory. Since strings are terminated by a zero value in the memory, we keep reading characters from the memory until we encounter the first zero value. The resulting string is printed on the console using `println`.

`Std_readInt` is implemented by using the `StdIn.readInt` function. After the value is retrieved from the console, it is pushed to the main stack.

`Std_readString` is implemented by receiving the input string by using `StdIn.readLine`. This string is stored in the data memory. In order to store the string in, WebAssembly code is generated by using the `Utils.mkString` function and then executing this code. The memory address for this string is pushed to the main stack after execution.

Getter and Setter Instructions

These instructions consist of `GetGlobal`, `SetGlobal`, `GetLocal` and `SetLocal` instructions and they are used in order to interact with the global and local memories. These instructions consume the value on the main stack and set the value of the given local to this value. The index of the local is part of the instruction. Getters and Setters for locals modify the call stack. Since other data is also stored in each frame, an offset is added to

the given address before accessing to the memory location. The program counter is simply incremented by one after the execution of these instructions.

Load and Store Instructions

In order to interact with the data memory the instructions `Store`, `Store8`, `Load` and `Load8_u` are used. Instructions that store a new value in the data memory consume two values from the main stack, the address of data memory and the value to be stored, whereas the instructions that load a value from data memory to main stack consume only one value, a pointer to data memory. After execution, the program counter is incremented by one and the next instruction is fetched.

The `Store` instruction stores a value to the data memory. Since the data memory is byte addressable, this operation is done by separating the `i32` value to four bytes. This is performed by using byte masking and shifting operations. After separation, these byte values are stored into four different memory locations starting from the given address. Bytes are stored in a little endian manner, the most significant byte being placed in the biggest memory address. The conversion from integers to bytes results in signed values, but this case is handled in `Load` operations.

The `Store_8` instruction works in a very similar way to the `Store` instruction. The only difference is that it stores a single byte instead of four bytes. Since it still consumes a value from the main stack, the value has to be masked and converted to one single byte. In this process, only the least significant byte of the input integer is used and stored to the given byte address in the data memory.

The `Load` instruction is used to load a value from the data memory to main stack. It consumes one value from the main stack, which is the address of the data memory location to be accessed. Since the data memory is byte addressed, the four consecutive bytes starting from the given address are accessed. Since these bytes are stored as signed bytes, we convert them to unsigned bytes by masking them after retrieval. These bytes are shifted to the left depending on their significance and summed to calculate the value that is to be loaded. The result of the calculation is then pushed to the main stack.

The `Load8_u` instruction works in a very similar manner to the `Load` operation. Instead of loading four different byte values, it loads only one byte from the given memory address and zero extends it to get an integer value. This integer value is pushed to the main stack.

4. Possible Extensions

Since the Virtual Machine that is developed for this project only deals with WebAssembly code generated by our compiler, it only receives a restricted set of WebAssembly instructions as input. Therefore, the number of instructions it is able to handle is also restricted. A possible extension can be adding more instructions from the WebAssembly specification to our Virtual Machine so that it can execute any kind of WebAssembly program.

References

Wikipedia contributors. Virtual Machine, 2017. URL `https://en.wikipedia.org/wiki/Virtual_machine`. [Online; accessed 11-January-2018].

Wikipedia contributors. Web Assembly, 2018. URL `https://en.wikipedia.org/wiki/WebAssembly`. [Online; accessed 11-January-2018].