# Virtual Machine for Amy Compiler

CS320 - Compiler Language Processing '19 Final Report

Berke Egeli Pinar Ayaz Efe Acer
EPFL
{berke.egeli, pinar.ayaz, efe.acer}@epfl.ch

## 1. Introduction

The aim of our project is to extend the Amy Compiler pipeline by adding a Virtual Machine that can execute WebAssembly code. This grants us the flexibility to be able to use the Amy Compiler regardless of the specific platform, since the Virtual Machine abstracts away the architecture dependent details.

The pipeline mentioned above consists of the following stages together with basic descriptions:

- **Lexer**: Groups characters into Tokens.
- **Parser**: Constructs a Parse Tree from Tokens according to the rules of the Amy Language. Transforms this Parse Tree into an Abstract Syntax Tree (AST) and removes unnecessary Tokens.
- **Name Analyzer**: Checks the correctness of the variable naming and assigns unique identifiers to every variable.
- **Type Checker**: Makes sure the typing rules of the Amy Language are satisfied.
- **Code Generation**: Generates WebAssembly equivalent of the Amy code.
- **Code Printer**: A follow-up stage for Code Generation in which the WebAssembly code is written into a file and outputs the Module object created by the Code Generation stage.

, to which we added the extra stage:

- **Virtual Machine**: Gets the Module object output of the previous stage and executes this WebAssemly representation.

In the Virtual Machine stage, where the execution takes place, the Machine itself is modelled using arrays. Each array corresponds to different hardware components that includes Main Stack, Call Stack, Data Memory, Instruction Memory and Global Memory. Pointers are used to reference various addresses in those components and thus control the flow of execution. This generic and fairly simple architecture allows us to achieve our goal of platform independent, flexible execution.

## 2. Examples

The execution of the compiled programs requires nodejs support and is triggered by the explicit node command. With our extension the execution can be handled by simply running the Main.scala file.

We expect no change in the output of any particular Amy Program whether it is executed using the Amy Interpreter, explicit node command or our Virtual Machine. An example is provided below:

Fibonacci.sc contains the following code:

```
object Fibonacci {
    def fib(n: Int): Int = {
        if (n < 0) {
            error("Invalid Input: fib: "
                ++ "n cannot be negative")
        } else {
            if (n == 0 || n == 1) { n }
            else { fib(n - 1) + fib(n - 2) }
        }
    }

    Std.printString("This program computes the nth "
        ++ "fibonacci number recursively.");
    Std.printString("Enter the value of n: ");
    val n: Int = Std.readInt();
    val nthFibonacciNumber: Int = fib(n);
    Std.printString("The value of nth fibonacci number is: "
        ++ Std.intToString(nthFibonacciNumber))
}
```

The execution of Fibonacci.sc:

```
sbt:amyc> run library/Std.scala examples/Fibonacci.sc
[info] Running amyc.Main library/Std.scala
examples/Fibonacci.sc
[ Info ] Grammar is in LL1
-> This program computes the nth fibonacci number
recursively.
-> Enter the value of n:
<- 10
-> The value of nth fibonacci number is: 55
[success] Total time: 7 s, completed 02-Jan-2019 16:37:59
```

## 3. Implementation

### 3.1 Theoretical Background

WebAssembly is a specification for an assembly language and a corresponding machine language that is supposed to run on a virtual machine in web browsers. In other words, WebAssembly is a new portable, size- and load-time-efficient format suitable for compilation to the web. It is designed to be called from JavaScript in browsers and allows high performance execution. Our Virtual Machine designed to run WebAssembly representations of Amy programs. [LAR]

A Virtual Machine is essentially a software program, which exhibits the behavior of a real computer and emulates the execution of this particular computer. The advantage introduced by using a Virtual Machine is to run programs independent of the host's platform.

Typically each computer architecture requires its own Virtual Machine for specific programming languages. However, since the Amy Language is built on top of Scala, our target machine is the Java Virtual Machine. This allows us to write a single Virtual Machine that can work in any architecture. [TEC]

### 3.2 Implementation Details

We modelled the architecture of our Virtual Machine using generic hardware components. Those components and their descriptions are given below:

#### 3.2.1 Architecture of the Virtual Machine

**Call Stack**

The architecture needs to know about the return addresses of the functions, which are called but not yet returned. The call stack is used for this purpose so it stores the return addresses of these functions together with additional information. The return address and the additional information composes a data segment. The zeroth place in the segments contains the return address, first place contains the number of locals in the function caller and the second place contains the number of locals in the function. We model it again as a Scala array of integers with a limited size of 4000000 entries. We hold a pointer to the call stack and model it an Int variable. This pointer refers to the return address of the latest called function that is not yet returned.

**Main Stack**

WebAssembly instructions use the Main Stack as the working stack. Main Stack simply holds information about WebAssembly instructions. We modeled it as a Scala array of integers, that has a fixed size of 4000000. A pointer is used to keep track of the top of the stack, which is the first unused slot of the stack. This pointer is modeled as a Int variable.

**Data Memory**

Program objects are stored in the data memory. Data memory is addressed using four byte words that are manipulated in little endian byte order, which means the least significant byte of data is placed at the byte with the smallest address. We modelled the Data Memory as a Scala array of bytes with a limited size of 4000000 entries.

**Instruction Memory**

Instruction Memory consists of the Amy programs. How large the programs are affects the size of the Instruction Memory. The program counter, a pointer, references the instruction to be executed, hence this pointer determines the flow of execution. We modeled the Instruction Memory as a Scala array of instruction objects with the corresponding Amy Program and its dependencies' size. The program counter is also modeled in software using a simple Int variable.

**Global Memory**

The global memory, as its name implies, is used to store global variables. It is modeled again as a Scala

array of fixed size, 10.

### 3.2.2 Executing the Code

Some amount of pre-processing is necessary for the Virtual Machine to execute the code. This pre-processing comes in the form of re-ordering the code and inserting additional lines. Since, the main method must be the first block of execution we move it to the beginning guranteeing that it is the first function in the program. Then, we add explicit return statements at the end of function to indicate the return address. After these operations, the instruction memory is populated with the pre-processed code instruction by instruction. Additionaly, the program counter is set to zero, to signal that the execution is in the starting state.

The Virtual Machine starts its execution by digging into the main method. The instructions pointed by the program counter are executed in order. Each instruction updates the value of the program counter to proceed with the control flow. Typically, executing an instruction adds one to the current value of the program counter, meaning that the next instruction to execute is the one that directly follows the current instruction. However, some instructions such as Branch and Call updates the program counter in their own way, since these instructions trigger different parts of the code and continue the execution from those parts. Hence, Branch and Call instructions use index maps to fetch the addresses of the instructions they want to jump.

We grouped the instructions into certain types that have their own way of being executed. These instruction types are given together with their descriptions as follows:

#### Accessor & Mutator Instructions

*Instruction Set* := { `GetGlobal`, `SetGlobal`, `GetLocal`, `SetLocal`}

These instructions are simple getters and setters that do not change the order of execution, meaning that the value of program counter is updated by one after their execution. The purpose of these instructions is to access or modify the variables. These are done by either getting the topmost value in the main stack and setting the variable to this value or updating the value of the variable to the main stack.

#### Load & Store Instrcutions

*Instruction Set* := {`Store`, `Load`, `Store8`, `Load8_u`}

Load and store instructions, again, do not change the order of execution, so they only increment the program counter by one. The goal of these instructions is to store values in the data memory or load values from it. This is accomplished in two different ways. Store instuctions get the two topmost values of the main stack, which correnpond to the store address in the data memory and the actual value to be stored. The value that will be stored is an i32 value, thus it is seperated into four bytes and stored in little endian order to fit our four byte word architecture. On the other hand, load instructions only get the topmost value of the main stack that corresponds to the loading address. The consecutive bytes covered by a word starting with the loading address are loaded, then converted to unsigned representation and the resulting value is pushed on top of the main stack.

The above description applies for the `Store` and `Load` instructions, `Store8` and `Load8_u` introduces an insignificant difference. That is, they operate on single bytes instead of single words.

#### Control Instructions

Control instructions change the control flow and their mere purpose is to do so. They update the value of the program counter in their own specific ways.

If instructions are arguably the most used control instructions in programming. They are used to navigate the control flow depending on a conditional value. When executing these instructions, we take this conditional value from the top of the stack and evaluate its value. Depending on the value we execute the if block (if the value is 0) or the else block (otherwise). A challenge is to determine the exact locations of the `Else` and `End` lines so that the intended blocks can be executed. This is done by exploring the whole if construct using another method.

Call and Return instructions are yet another type of control instructions. These instructions are used, when

the programmer wants to call another function from the code base. When the call instuction is executed, we populate the call stack with the particular return address and the number of local variables of the callee method. Then, we push these local variables to the main stack. After these, the callee becomes able to easily access the caller related information from the call stack and get its inputs from the main stack. The execution of the return instruction is relatively simple, since it trivially updates the program counter to take on the return address stored in the call stack. The Amy I/O functions are specific applications of the generic Call instructions, since these functions depend on Scala I/O libraries we implemented their execution by navigating the control flow to the corresponding Scala functions.

Branch and Loop instructions are another part of control instructions. These instructions specify certain labels that indicate the point in the code where the program counter will jump. We store the relation between the labels and the locations they indicate in a Scala map. The execution of the Branch and Loop instructions are simply done by fetching the value of the specified label from the map and updating the program counter to take on this value. A similar map is also used for if instructions to prevent exploring already explored jumping points.

### Logical & Numerical Instructions

Logical and Numerical instructions are simple instructions that do not change the order of execution. In other words, they increment the value of program counter by one after being executed. Their use is to perform a logical or an arithmetic operation on two operands. These operands are obtained from the main stack. The two topmost items are popped from the stack and the specific operation is applied to them, then the result is pushed back to the stack.

## 4. Possible Extensions

A possible extension we could have made is to separate the Virtual Machine stage from the regular Compiler pipeline. This would prevent us from recompiling the code each time it gets executed, which in turn makes it more efficient to execute portable code.

We could not figure out how to separately compile the program and run the program due to time constraints. We believe that we can do this by defining a new Main method that does not run each stage of the pipeline individually but instead use some stored outputs.

## References


Code generation. `http://lara.epfl.ch/w/cc18:labs`$_0$6.$Accesse$ $2019 - 01 - 02$.

What is a virtual machine (vm)? - definition from techopedia. `https://www.techopedia.com/definition/4805/virtual-` Accessed: 2019-01-02.