CS4215

TEAM PONTEVEDRA

Source to EVM Compiler

Authors:
Yuchen WANG (A0200093X)
YiJia CHEN (A0200138X)

NUS School of Computing

April 17, 2022

Contents

1 User-level Documentation

source2evm is a Source to EVM bytecode compiler written in Typescript. This project comes bundled with standalone EVM binaries for both `arm64` and `x64` architecture that can be used to execute compiled code.

The compiler currently supports the compilation of a subset of Source to bytecode for the Ethereum Virtual Machine (EVM). Syntax follows that of Source.

1.1 Installation

Compiler release can be downloaded from here on our GitHub repository

To build:

```
yarn install
yarn tsc
```

1.2 Usage

The compiler can be used through the `NodeJS` script `source2evm.js`. If executed without compiled compiler code, the script will compile the compiler with `tsc`.

```
./source2evm -i [input file] [option]

Options:
    -r: Execute the code with bundled EVM after compilation.
    -o: Write output bytecode to output file.
    -h: Show help message.
```

1.3 Known Issues

Please switch to Node v14 if there are any issues with running the compiler. Newer versions of Node could cause problems with dependencies.

1.4 Supported Features

- Integer arithmetic
- Boolean operations
- Declaration of variables and constants
- Functions
  - Function declarations and applications of named and anonymous functions
  - Nested functions
  - Recursion, tail call optimisation, mutual recursion
  - Functions as parameters and return values
- Conditionals

  - Ternary operator & if-else statements

- While and for loops
  - Note that `break` and `continue` are not supported

1.5 Test cases

A number of test programs can be found under `/tests`. The expected result of evaluation can be found at the end of each file.

1.6 Things to note

1.6.1 Output

Similar to the Source interpreter, the compiled code will always return the result of the last statement of the given program that has return results. However, unlike the Source interpreter, if none of the statements in the program have return results, the code will return 0.

1.6.2 Unused return results

Due to the compiler's reliance on the EVM's stack for exiting from functions, **any** unused return values from any statements **within functions** will cause undefined behaviour, and will likely lead to EVM errors. Such statements can still be used outside of functions.

1.6.3 Scoping

Note that ALL arguments are passed-by-value to functions, hence variables are NOT mutable across function scopes.

```
let y = 1;
function f() { y = 2; return 0 };
y;
```

The above code will return 1, as `f` only changed the value of the copy of `y` within its own scope.

1.7 Compiler Error Messages

There is no static type checking in the compiler, but the compiler will check and throw exceptions for the following:

- Reassigning values to constants

  ```
  const x = 2;    x = 3; // reassigning const, compiler will throw exception here
  ```

- Referring to undeclared name

  ```
  y + 4; // y not declared, compiler will throw exception here
  ```

- Using an unknown operator

  ```
  1 $ 2; // $ is not a supported operator
  ```

Note that there will not be any line number information in the error messages.

2 Developer-level Documentation

2.1 Operators

The following operators are supported:

Operator	Type of operand 1	Type of operand 2	Return type
+	Number	Number	Number
-	Number	Number	Number
*	Number	Number	Number
/	Number	Number	Number
===	Number/Boolean	Number/Boolean	Boolean
<	Number	Number	Boolean
<=	Number	Number	Boolean
>	Number	Number	Boolean
>=	Number	Number	Boolean
&&	Boolean	Boolean	Boolean
||	Boolean	Boolean	Boolean
!	Boolean	-	Boolean

The ternary conditional operator is also supported:
`condition ? if-true : if-false`

When a known number needs to be pushed onto the operand stack, `PUSH32` is always used to push it as a 256 bytes integer.

$$\frac{n \rightarrow i}{n \hookrightarrow PUSH32 \cdot i}$$

For most operators with equivalent EVM instructions, the translation is simply a mapping to the equivalent instructions. Note that for operators where the order of the operands matter, e.g. - and <, the order in which operands are computed and pushed onto the stack is flipped, where the second operand is pushed first, due to how EVM instructions operate on the stack.

$$\frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 + E_2 \hookrightarrow s_1 s_2 ADD} \qquad \frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 - E_2 \hookrightarrow s_2 s_1 SUB} \qquad \frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 * E_2 \hookrightarrow s_1 s_2 MUL}$$

$$\frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 / E_2 \hookrightarrow s_2 s_1 DIV} \qquad \frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 === E_2 \hookrightarrow s_1 s_2 EQ} \qquad \frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 < E_2 \hookrightarrow s_2 s_1 LT}$$

$$\frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 > E_2 \hookrightarrow s_2 s_1 GT} \qquad \frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 \&\& E_2 \hookrightarrow s_1 s_2 AND} \qquad \frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 || E_2 \hookrightarrow s_2 s_1 OR}$$

Operators without any equivalents are translated to a combination of instructions.

$$\frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 <= E_2 \hookrightarrow s_1 s_2 EQ \cdot s_2 s_1 LT \cdot OR} \qquad \frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2}{E_1 >= E_2 \hookrightarrow s_1 s_2 EQ \cdot s_2 s_1 GT \cdot OR}$$

<= and >= operators are simply replicated by replacing them with conjunctions of < or > with ===.

$$\frac{E_1 \hookrightarrow s_1}{!E_1 \hookrightarrow s_1 ISZERO}$$

While there is a `NOT` instruction in EVM, it is a bitwise `not` that inverts all 256 bytes of the input when EVM uses 0 and 1 for false and true respectively, hence to translate `!`, we check if the given operand is 0 using the `ISZERO` instruction. If a value is 0, it is originally treated as false in EVM, and `ISZERO` will return 1, which is true. If a value is any other non-zero value, `ISZERO` will return the false value of 0 instead. In either case, `ISZERO` returns a boolean value that is the logical negation of the given operand.

$$\frac{E_1 \hookrightarrow s_1 \quad E_2 \hookrightarrow s_2 \quad E_3 \hookrightarrow s_3 \quad compile\_conditional(E_1, E_2, E_3)}{E_1 ? E_2 : E_3 \hookrightarrow s_1 \cdot JDEST1\_PC \cdot JUMPI \cdot s_2 \cdot JDEST2\_PC \cdot JUMP \cdot JDEST1 \cdot s_3 \cdot JDEST2}$$

Where `JUMPI` takes the the top operand on stack as the destination to change *program counter (PC)* to, and jump if the second operand on stack is true. `JDEST` is short for `JUMPDEST`, the instruction that adds a label for jump destinations. Each `JDEST` in the above proof tree is labelled with a number behind for clarity. `JDESTi_PC` refers to the PC location of the corresponding `JUMPDEST` to be used as destination for `JUMP` and `JUMPI` instructions. They are calculated at runtime by first putting current PC on stack with the `PC` instruction, then adding the PC distance from that point to the destination, which is calculated during compilation.

2.2 Conditionals

The ternary conditional operator is documented above. Conditional expressions in the form

```
if (cond) {
    // if true
} else {
    // if false
}
```

are first converted to the ternary operator form, then their condition expression, truth expression, and false expression are extracted to be passed to the `compile_conditional` function to be compiled in the same way as the ternary operator version.

$$\frac{\texttt{if } (E_1)\ \{E_1\} \texttt{ else } \{E_3\}}{E_1 ? E_2 : E_3}$$

For if-else form of conditionals, the expressions within curly braces do not have their own independent scope.

2.3 Memory model

EVM does not have registers or a conventional runtime stack, hence we have decided to use the memory to simulate both a runtime stack and shared registers.

We have allocated the memory from 0x40 to 0x200 for stack frame pointers, and addresses from 0x220 onward are used for the call stack. Each stack frame on the call stack will contain values for all names that environment, which may either be a function or the main scope, contains within its lexical scope. As all names are allocated an address at the point of initialisation of that environment, which, for this compiler, would be at the during initialisation of functions.

To keep track of the current active stack frame, addresses 0x00 and 0x20 are used as registers. 0x00 always contains the address of the current stack frame pointer, i.e. the stack frame pointers stored from 0x20 to 0x200. 0x20 will always store the actual address of the current stack frame, i.e. the value held at the address 0x00 points to.

The stack frame pointers each point to the start of its corresponding stack frame. As the stack grows downwards, more stack frame pointer entries are placed onto the stack frame pointer stack, growing it with the call stack. For both the call stack and the stack frame pointer stack, the next available address is the one that immediately follows the current active stack frame. For the stack frame pointers, the next pointer will be pushed to the address of the current stack frame pointer plus 0x20. For the stack frame itself, the next available address will be the start of the current stack frame offset by the number of variables it contains times 0x20.

When a function returns and the stack frame is no longer needed, the exiting stack frame will be "popped" by pointing the active stack frame pointer to the previous stack frame, "moving up" in both the stack frame pointer stack and the call stack. When another function is called, it's stack frame will start from the address previously used by the exited function, overwriting the previous memory. Garbage collection is thus not needed as memory is implicitly reclaimed upon function exit.

2.3.1 Variables

During compilation, the compiler keeps a lookup table of the names known to each environment under `closure_lookup.locals` in the compiler, and each name is given an offset amount, starting from 0x20, and increasing by 0x20 with every name. These names are the variables that the program can access in that environment. To access a variable, EVM takes the value at 0x20, then adds the offset for that variable name, which was determined at compile time, to obtain the actual address that variable resides at, where it can then invoke `MLOAD` or `MSTORE` to read from or write to that variable.

2.4 Loops

Loops carry their own closures. Two jump destinations are created during compilation, one before the condition and one at the very end of the loop, before the start of the rest of the program. `JUMPI` is invoked to send the PC to the loop body, otherwise if the condition is not truthy, the PC will skip outside the loop.

2.4.1 While Loops

```
while (cond) {
    // body
}
```

2.4.2 For Loops

```
for (let i = 0; i < 0; i = i+1) {
    // body
}
```

For loops have an additional three-part sequence that also exist within the loop's closure. A minor tweak would be to treat the **updating statement** as part of the loop body, the **test statement** as the loop condition, and be very careful to exclude the **initialisation statement** during the jump.

2.5 Functions

Functions exist in two forms, the code compiled from their body, and the PC offset serving as entry points to jump to. Within the final compiled bytecode, the bytecodes for all functions, named or anonymous, are placed at the start. The PC offsets for each function is thus known at compile. During runtime, the named functions are treated in the same way as other constant variables, only that the value stored in memory for a function is its PC offset. This allows functions to be passed as parameters and returned as return values just like any other data.

2.5.1 Function definition

Functions can be defined in two ways, either through a function declaration:

```
function f(arg) {
    // function body
}
```

or through a constant declaration of a lambda expression:

```
const f = arg => {
    // function body
}
// or:
const f = arg => return statement
```

The function declaration form will be transformed to the constant declaration form. The constant declaration will be passed to the `compile_constant` function, which will extract the constant name, in this case the function name, and pass the lambda expression to the `compile_lambda_expression` function for code generation. The compiled bytecode for all functions will be placed near the start of the final compiled program. `compile_constant` will, in the case of functions, return the bytecode for pushing the PC offset, which is known at compile time since it is just the number of bytes from the start of the compiled program, for the `JUMPDEST` of the function entry. During runtime, this PC offset will be stored into the address allocated in the stack frame for that function name. When that name is accessed like any other names, the PC offset will be retrieved, which EVM will call `JUMP` on to enter the function.

`compile_lambda_expression` will construct the compiled code in three parts:

- *Read and store arguments*: This is the first part of the function code that is ran when the function code is accessed. Since the function code is only accessed during application, it is assumed that before the jump, the application in caller has already placed all required arguments for the function call on the operand stack. For every argument the function has, `compile_lambda_expression` will generate bytecode that takes the top element on operand stack, and stores it in memory, based on the stack frame offset for the argument name, determined at this stage based on its order of appearance in the given code, and the stack frame address, determined at runtime. `compile_lambda_expression` will also record names and assign offsets for all locally declared names in the compiler lookup table.
  - Note that compiler will scan for all free variables in the function and in all its nested functions, and these variable names will be added appended to the list of arguments. The original arguments will be read and stored first, followed by the free variables.
  - Original arguments are assigned offsets first, followed by free variables, then locally bound variables.
  - During application, input arguments are pushed to stack in the order that they are given, hence the last argument will be the first value on stack, so arguments are read from stack in reverse order.
- *Function body*: This is the main body of the function. The body of the function is extracted and passed to `compile_expression` with the updated lookup. The returned result will be the bytecode containing the main logic of the function.
- *Stack frame and PC updates* Every function application creates a new stack frame during runtime, and moves back to the caller's stack frame upon function exit, hence code for updating runtime stack at both the start and end of the compile function bytecode is needed. After the function is fully executed, PC also needs to be restored to after the `JUMP` invoked by function application.
  - Return PC is pushed on stack before arguments are pushed during function call. We expect functions to always have a single return value that is the result of the final statement executed in the function or a return statement. When the function is ready to return, the top two elements on operand stack should be *return value* and *return PC*. `SWAP1` and `JUMP` are then called in that sequence to jump to return PC and leave return value on top of the stack.
    - ∗ This approach is the reason for statements with unused return values to result in unexpected behaviours that would most like cause jump errors in EVM, as the presence of additional values on operand stack causes return `JUMP` to not jump to the correct PC.

The code generated for a function thus follows the following structure:

Jump Destination
Create new stack frame and update pointers to runtime stack
Get and store arguments from operand stack
Function body
Return to previous stack frame
`SWAP1` and `JUMP` to jump back to caller

2.5.2 Function application

When a function is applied, e.g. `f(1, 2, 3)`, the return PC is first calculated by calling the instruction `PC`, then adding the distance from there to 1 byte after the `JUMP` instruction used to enter the function. This puts return PC on stack, allowing PC to be set to the instruction that immediately follows the function application.

This section will only cover application of named functions, application of anonymous functions will be covered later.

When there is a function application, the compiler generates code for the application in the following steps:

- *Prepare captured variables*: compiler will retrieve list of captured variables from the lookup, then searches for them within the current scope, and generate code that load them from memory, pushing them onto stack.
- *Prepare arguments*: the actual list of arguments will now be compiled, which will result in their values being pushed on top of the operand stack at the end.
- *Get PC offset of function to be called*: compiler generates code that will retrieve, from memory, the PC offset of the function to be called. Such offsets are stored in memory like any other variables.
- *Prepare jump and return PC*: compiler will then append `JUMP`, code for returning stack frame, and `JUMPDEST` for the actual entry to function and eventual return. With the length of the code known now, the return PC can be calculated with `Current PC + length(code) + 1`. The return PC will be pushed on to stack first to put it below function arguments.
  - If the returned expression is a function, the function's free variables will be first pushed on to the operand stack before `JUMP` is called. The application of functions that return functions thus have to be followed by another application immediately to consume the function returned by calling `JUMP` with the returned function PC on the stack.

Function PC
Function arguments ⋮
Function captures ⋮
Return PC

Stack at the point of `JUMP`

2.5.3 Variable capture

Instead of extending the current closure to capture variables from outer scopes, this compiler extends the list of arguments instead. The function `scan_out_names` scans out all names in a given parse tree that are *not* locally declared or, in the case of functions, one of the function arguments. This scanning is done at the start of a sequence of expressions before any functions are actually compiled using the `scan_out_names` function through DFS of the entire parse tree. The resultant list of list of captured variables for each function is stored in the lookup table under `funcs`, with each list mapped to the name of the function it came from. This list is referred to when compiling both lambda expressions and applications.

In addition to free variables, the name of the applied function is also added as a parameter. This is to allow for nested recursions like the following to be possible.

```
function f() {
    return g();
}
function g() {
    return f();
}
```

2.5.4 Tail call optimisation

For functions whose return expression consists of only a function application, the compiler compiles the return differently through the `compile_tail_call_recursion` function. During runtime, this application will first pop the current stack frame before entering the new function, which generates another frame.

2.5.5 Anonymous Functions

Anonymous functions are defined when they are applied, or as return results (in which case they need to be applied immediately after being returned). They are thus compiled during application. When compiling a return expression or application and the expression is a lambda expression, they are compiled as lambda expressions and have their free variables scanned at that moment. Their PC offset is not stored in memory as it is only used once. They behave similarly to named functions other than when they are compiled.

2.6 Compiler functions

2.6.1 `compile_expression`

Applies the appropriate compiler function depending the type of expression.

$$\frac{E}{E \hookrightarrow s}$$

2.6.2 `compile_sequence`

This function takes a sequence of expressions and does two things:

Convert free variables in each function to arguments

1. Scans out all functions in the program through DFS
2. For each function, retrieve name and scan out all free variables, including those in nested functions
3. Store, in `closure_lookup.funcs`, the function name to captured variable pairs, pushing in the name of the function itself

Compiles every expression in the sequence by applying `compile_expression`

$$\frac{E_1 \hookrightarrow s_1, ..., E_n \hookrightarrow s_n}{\{E_1, ..., E_n\} \hookrightarrow s_1 ... s_n}$$

2.6.3 `compile_constant`

1. Get constant name
2. Record name in `closure_lookup.locals`
3. Add name to in `closure_lookup.constants`
4. If is a function, call `compile_lambda_expressions`
5. Generate code for pushing value (or function PC offset) to operand stack and storing in appropriate location

2.6.4 `compile_conditional`

$$\frac{E \quad E_1 \quad E_2}{E \; ? \; E_1 : E_2} \qquad \frac{\Delta \Vdash E \rightarrowtail true \quad \Delta \Vdash E_1 \rightarrowtail v_1}{\Delta \Vdash E \; ? \; E_1 : E_2 \rightarrowtail v_1} \qquad \frac{\Delta \Vdash E \rightarrowtail false \quad \Delta \Vdash E_2 \rightarrowtail v_2}{\Delta \Vdash E \; ? \; E_1 : E_2 \rightarrowtail v_2}$$

2.6.5 `compile_while_loop`

Creates a `new_env` that extends from its enclosing environment. All statements within the loop operate as per usual through `compile_expression` within `new_env`. PC offsets are generated on compile time to demarcate the start of condition checking and end of loop, respectively.

2.6.6 `compile_for_loop`

Same as while loop, except for special jump instructions to accommodate the three-part initialization sequence.

2.6.7 `compile_lambda_expression`

Generates code as follows:

Allocate space on stack and consume arguments on operand stack

$$\frac{(x_1,...,x_n,v_1,...,v_m) => \{E\}}{(x_1,...,x_i,...,x_n,v_1,...,v_m) \hookrightarrow (PUSH4 \cdot 32i \cdot PUSH \cdot 0x20 \cdot ADD \cdot MSTORE)^{(n+m)}}$$

where $x^{(n)} := x$ is repeated n times

Compile lambda body

$$\frac{E \hookrightarrow s}{(x_1,...,x_i,...,x_n,v_1,...,v_m) => E \hookrightarrow s}$$

Return PC

$$\frac{}{SWAP1 \cdot JUMP}$$

The resultant chunks of code are concatenated together, prepended with `JUMPDEST`, and appended to the global variable `constants`. The code for storing PC offset of this function to memory is returned.

2.6.8 `compile_application`

1. Calculate return PC and push onto operand stack
2. Compile arguments and captured variables

   $$\frac{E_1 \hookrightarrow s_1,...,E_n \hookrightarrow s_n, v_1 \hookrightarrow c_1,...,v_m \hookrightarrow c_m}{(E_1,...,E_n,v_1,...,v_m) \hookrightarrow s_1...s_n v_1...v_m},$$

   where v_i are captured variables of the function.
3. Generate code to retrieve function offset
4. Append `JUMP` and `JUMPDEST`

2.6.9 `compile_tail_call_recursion`

After computing arguments and pushing them onto stack, retrieve PC offset for applied function, then pop current stack frame before pushing return PC and calling `JUMP`.

2.7 Known Issues

1. Loops cannot accommodate function calls within its scope as the PC offsets of the function block is not easily obtainable through the existing implementation. Better pre-processing of functions and loops would certainly resolve this issue.