Cmpe492 Final Progress Report

Yiğit Özkavcı
2013400111
yigit.ozkavci@boun.edu.tr

June 2018

Contents

1 Introduction **2**

2 Dynamic Memory **2**
2.1 MEMORY 2
2.2 STORAGE 2

3 Dynamic Data Structures **3**
3.1 Runtime Representation 3
3.2 Arrays 3
3.2.1 Resizing Arrays 4
3.3 Hashmaps 4

4 New Language Features **5**
4.1 Runtime Exceptions (Halting) 5
4.2 Logging 6

5 Compiler Features **6**
5.1 Testing 6
5.2 Showing AST 7
5.3 Showing Bytecode 7
5.4 Showing Disasm 7
5.5 ABI Module 7

6 Conclusion **8**

1 Introduction

This is the CMPE492 final progress report for project Ivy, a programming language for writing smart contracts on Ethereum Virtual Machine (will be referred as EVM from now on).

This report represents the conclusion of the 1-year progress of Ivy.

This term's work focused on runtime representation of various newly-introduced data structures such as dynamic arrays and hashmaps. Aside from dynamic data structures, we also improved our development workflow, introduced several test scenarios.

2 Dynamic Memory

When it comes to representing dynamic data structures, EVM actually is not so different than other virtual machines. We are still able to split memory in stack and heap parts, on which stack represents static allocations and heap represents dynamic ones.

The only differing part of EVM in this manner is that EVM has two separate ways of storing things in memory: MEMORY and STORAGE. MEMORY represents temporary storage, which means that whatever we store in MEMORY, we will have them wiped out after our function call terminates. This corresponds to local variables in programming; whereas STORAGE represents persistent storage, whatever is stored here, persist through the lifetime of the contract on blockchain.

There are more details on how MEMORY and STORAGE works under the hood, and we dive into their details in the following subsections.

2.1 MEMORY

MEMORY is the temporary memory for ethereum smart contracts. It's cheap to use incrementally, but there is an important detail in using memory: when a contract is deployed, the farthest location we allocate is considered the memory limit we allocate, hence we need to make use of memory very carefully. So if we were to only allocate memory location 0x2460, contract is said to allocate 0x2460 x 0x20 bytes of memory, so it's crucial that we make use of every part until the maximum point of memory allocation.

2.2 STORAGE

Storage is the persistent memory for ethereum smart contracts. We represent storage varibables as global variables in Ivy. Storage is generally very expensive to use, but for it's also a much better way to store hashmaps because of how MEMORY works in EVM. (see 2.1). We will discuss how we store hashmaps and compute hashing function later in section 3.3

3 Dynamic Data Structures

3.1 Runtime Representation

In order to represent dynamic data structures, we first needed to change our representation of operands, which correspond to mostly right-hand side in the context of Ivy. Until now, we evaluated our operands, stored them then got their addresses in compile-time. But as you might imaginge, it's impossible to know address of a runtime variable at compile time. Because of this limitation, we started no longer storing operand addresses, but instead, have their values in our stack. So for a statement like this: $int\ a = 5;$, we first evaluate 5, which is $EInt\ 5$, we push $0x05$ to stack then return; after having evaluated the operand, we store the first element of stack (which is 5 in this case) in the address of a. This part is crucial: we always store variable addresses in our lookup table, but for the operands, we may or may not know the address, so they are represented in stack instead.

3.2 Arrays

Representing arrays in heap space required planning since we needed a well-defined structure for storage. We decided on the following pattern:

length	element 1	element 2	...

But where do we put this? We know that array identifier's address should be static, so when user types $int[]\ arr$, we need to determine a stack address for this variable, and it should not change. We concluded that stack address should point to a "variable heap address". This means that we know how to access array contents, but we don't know the actual destination of it.

Values	arr-addr	>...<	length	element 1	element 2	...
Addresses	stack-addr	>...<	arr-addr	arr-addr + 0x20	arr-addr + 0x40	...
Spaces	Stack Space	>...<	Heap Space			

Above figure represents how we store an array in heap space. The value in stack address acts like a pointer to the address where we store array in heap. When we go to that address, we have the array size, and then we can safely traverse the whole array.

3.2.1 Resizing Arrays

Resizing arrays involved more EVM assembly code than we expected, and it turns out that it was because it involed both reading and storing chunk of arrays.
Task of resizing arrays branches into 2 distinct parts: when we want to shrink it, or expand it. Shrinking is easy: we just set the size of the array to a smaller value; this lets us treat this array with the new length from now on.

The more difficult part is the expanding part while resizing arrays, because we need to find a larger place for the array since we don't have the guarantee that array will reside in the edge all the time, obviously. So we first started the procedure of allocating an array with the same length, and inside a for loop (which are just jumps in EVM assembly) we carried elements one by one. Of course, we do this in runtime with a loop, not by pasting more and more instructions since it would result in a larger bytecode.

The new space in the array is only a bunch of zeros (0x00). We don't throw any error in case user wants to access these places, even though we might throw errors in future. For more on exceptions and logging, see 4.1 and 4.2

3.3 Hashmaps

There are not enough information on how to store hashmaps on EVM, but we made use of this great article [1] in order to accomplish this task. It turns out that EVM actually is quite supportive when it comes to store key-value data, but with one small detail: we can do it only in STORAGE. We will come to the reason in this section, but after we discussed computing the hash function.

Computing the hash function is not a very complicated task since we have the necessary tools: a key to differentiate each hashmap and their keys, and a hashing function. We will use keccak256 hashing, which is accomplished in runtime via $SHA3$ instruction of EVM. We absolutely need to do this in runtime because we may or may not know the key to the hashing function. But how do we find the suitable key?

We have two components as the key to the hashing function: key of the mapping identifier (for $myMap[\text{"}foo\text{"}]$, it's "$foo$"), and something to uniquely identify our maps: ordering for the identifiers (ie. $myMap$). For each mapping identifier, we have a integer as ordering that starts from zero and increments by one. So if we have hashmaps $map1$, $map2$, $map3$, ordering of $map2$ is 1. Then we concatenate these two values, and we have our key to the hashing function. Below, we present some examples and their hashing results:

Variable	Identifier	Key	Hashing Function
$myMap[\text{"}name\text{"}]$	myMap	name	SHA3(order(myMap) ++ name)
$secondMap[\text{"}key2\text{"}]$	secondMap	key2	SHA3(order(secondMap) ++ key2)

Now that we have our hashing function, we can start storing and reading hashmap values. We use result of the hashing function as the address to our values, but these adresses are weird and random-looking values, and if we are to use them as addresses, we are going to have a very discrete scheme of values on our memory. As we discussed how MEMORY and STORAGE structured, and with MEMORY we were assumed to have allocated every cell until our latest allocation, allocating the addressed that are resulted from hashing functions are obviously infeasible since we would end up with allocating massive amounts of memory for no reason.

This is actually the very limitation Solidity still has: noone can use hashmaps in local scope with Solidity, since local variables correspond to MEMORY, in other words, the temporary storage in solidity. Hence, we also allow users to declare (and define) their hashmaps only in global scope.

Not being able to allocate hashmaps in local scope might be a limitation that we can overcome in the future, but it surely is going to alter the way we store our hashmap values.

4 New Language Features

4.1 Runtime Exceptions (Halting)

With the coming of runtime data structures, the concept of runtime exceptions arose. If someone tries to reach to a non-existing index of an array, we can currently detect it since we keep the length information in heap, and we have the indexing operand at hand. With $LT, GTandEQ$ instructions, we can compare the index with the existing length and determine whether someone tries to access an out of bound index of an array. In this case, we print a log message (event in Solidity terminology) and halt the execution, hence ending the transaction.

If we don't control this scenario, user was going to receive the value 0x00 without noticing s/he actually made a mistake; this is a case we want to avoid since in the philosophy of Ivy, we have safety in mind.

In the case of hashmaps and accessing a non-existing key, we don't check it since smart contract developers are used to Solidity conventions, which keeps saying "when a hashmap is created, every value is filled with zeros", hence no access violate the hashmap lookup. This state might change in future.

4.2 Logging

EVM provides several instructions for working with logging, and they are surprisingly rich in terms of categorisation. We can optionally include topics in our logs, and we can put anything we want as topics (limitation is 0x20/32 bytes). For the contents of logs, we don't put the contents of logs to stack as an input to LOG instructions, but instead, we put the address limits of the content. So if we want to log some content, we need to have it in memory, and specify its beginning and ending address as an input to instruction. This is actually a very clever way of representing can-be-large content since we can use up to 32 bytes as address.

With out javascript client which makes our deployment easy for us, we currently collect logs from every running Ivy transaction and display it. This is a **very** helpful way of debugging programs, since we now have a way of seeing actual runtime values for any Ivy programs, which would have been impossible without the logs since runtime is theoretically in any computer.

5 Compiler Features

5.1 Testing

In the Spring 2018 term, we greatly improved our testing suite. We currently have 13 testing scenarios and for each one we start a test network, deploy our contract and run the transaction then collect and assert the result. It's currently very easy to write tests on ivy: one just needs to modify *test/assertions* file and it's good to go. The file currently has the following structure:

```
#!/bin/bash

ASSERTIONS=( Hashmaps.ivy "9" "string_keys()"
             Hashmaps.ivy "28" "integer_keys()"
             Resizing.ivy "5" "read()"
             Resizing.ivy "98" "write(98)"
             Resizing.ivy "0" "out_of_bounds_read()"
             Resizing.ivy "0" "out_of_bounds_write()"
             DynArr.ivy "9" "main()"
             Id.ivy "3" "id(3)"
             Fibonacci.ivy "610" "fib(15)"
             Adder.ivy "10" "add(3, 7)"
             Adder.ivy "3" "add(1, 2)"
             Branching.ivy "3" "main(4)"
             Branching.ivy "6" "main(5)"
           )
LEN=${#ASSERTIONS[@]}
```

5.2 Showing AST

Ivy compiler can also perform other tasks than actually compiling and running tests. It can show only the Abstract Syntax Tree of the ivy program given as input. This is a powerful feature that we have for free when we modulate the compiling pipeline good enough. As we described in older reports, we have the following pipeline: $lexing -> parsing -> code\ generation$.

5.3 Showing Bytecode

After generating the bytecode necessary for EVM to run, we can display it without running it; but one has to be careful: generating the bytecode requires the actually code generation phase, which means code will be wholly evaluated and type-checked before we can evaluate the code.

The generated bytecode then can be safely used to deploy the contract with accompanied ABI (see 5.5).

5.4 Showing Disasm

Another important feature of Ivy compiler is that we can make it show the EVM assembly code that will be generated for a Ivy source. This is especially helpful to debug smaller programs, but it's increasingly difficult to debug larger programs since potentially assembly code can grow into 6000+ lines of assembly.

5.5 ABI Module

ABI stands for Application Binary Interface, which is a specification of callable functions and their input/outputs in json format. In order to make deployment painless, a good smart contract compiler should be able to generate ABI for any of its program, and Ivy can do that. With modules $Ivy.ABI$ and $Ivy.AbiBridge$, we can transform our AST to ABI and json-encode it before deployment. Just before every deployment, Ivy compiler encodes the abi for given program and substitutes a pre-decided string (@abi@ in our case) in our deployment script. This is currently a dirty way of handling deployment, but it does its job and we've already written several smart contracts this way.

Being able to transform AST to ABI with just one pure function and being able to collect errors is a powerful feature we use thanks to Haskell language. In fact, in the $Ivy.AbiBridge$ module, we don't even know the details of how to convert an AST to ABI, we just call the $Ivy.ABI$ module's function and that's it.

6 Conclusion

This report concludes the development of Ivy as a senior project in Boğaziçi University. Development of Ivy will continue, and we can see Ivy as a viable alternative to other smart contract languages out there in the future.

Future plans for Ivy includes the language IvyIR. We plan the current Ivy to be a intermediate representation (IR) language, and a new language called Ivy will be built upon IvyIR as a high level pure functional language. This way, we continue developing Ivy and IvyIR in parallel, with Ivy facing towards the user, and IvyIR getting more and more low-level, and ideally, we stop the development of IvyIR and only continue developing Ivy since the only mission of the IvyIR will be to be an imperative bridge between EVM assembly and Ivy. For this very task, we wanted to keep Ivy's syntax very lightweight and c-like since the compiler will be the one who generates code in this language.

References

[1] Are Ethereum Contracts Vulnerable to Hash Table Poisoning Attacks?: https://hackernoon.com/are-ethereum-contracts-vulnerable-to-hash-table-poisoning-attacks-a4d9241e16c4