Cmpe492 Midterm Progress Report

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

March 2018

Contents

1 Introduction

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

First part of the Ivy was developed in the course of CMPE491, for which a midterm and final report already exists and can be found here[1] and here[2].

In this term, we focused on language improvements targeting a more simplified memory model, deployment of our bytecode into blockchain networks and testing this behavior. We will illustrate each of these improvements/features in their own sections.

2 A New Memory Model

We used to have a pointed memory model, which was essentially allocating each variable according to its type. While working on deployment of the Ivy, we realized that memory operations should be optimized according to time, not space, which causes faster operations, but more space to be allocated. To see why can't we have best of the both worlds, let's look at EVM's memory-related instructions which are highly well-documented in the yellowpaper[3]:

```
MLOAD: Load a word from memory
MSTORE: Save a word to memory
MSTORE8: Save a byte to memory
```

Instructions above show that if we want to load data from memory, we can only load a whole 32-byte (*word* above means 256-bit, in other words, 32 bytes in the context of EVM), then crop out the parts we are not interested in. As for saving, we have more option. We can save a whole word, or a single byte to memory. It's fine for, for instance boolean types, but if we have a 8-byte integer, which corresponds to int64_t type in C, we have two options:
We can either save it byte by byte, or we can save a whole 32-byte, which overwrites the whole 32-byte cell in memory hence should not be the way to go.

This is actually the root of our problem: if we want to be efficient in terms of space, we have to save our types byte by byte, which causes us to run n instructions to save a n byte data structure. As we know, running instructions means *gas* cost in a smart contract, so we should always be minimizing the instruction count to make sure user does not spend more than what she should. This is why we felt urge to simplify our memory algorithm and make it less space efficient, and more gas efficient.

So we decided to take a third way: save every variable in its own 32-byte cell, and just use **MLOAD** and **MSTORE** instructions for now. This lets us a greatly simplified memory model on which we can easily develop optimizations. We basically keep an index to the end of our memory, from which we can search a new empty space to allocate. We still have pointer, but a single one now, for memory cells. When we want to allocate space for a variable, we begin searching from this pointer to infinity (notion of infinity modeled using lazy evaluation in Haskell, described in section 5.1.2), and if we find an empty cell, which we surely expect to do, we allocate it and save it to our lookup table.

3 Phases of a Smart Contract Bytecode

A smart contract's lifecycle consists of two main parts called **init** and **code**. When a smart contract is deployed, of course not all of it's code for functions are executed; a smart contract in its heart is essentially a list of functions and a top level switch-case statement to look for which function to call at that transaction. **init** phase can be thought of as the switch part, and **code** phase as the case statements.

This is indeed a simplified explanation, but it helps us understand what we are dealing with when we say "phase". Following two sections will illustrate how we implement init **init** and **code** phases, as we need to get into their details.

3.1 Init Phase

This phase is the entrance point of a contract deployment, even though we are going to explain deployment with depth in section 4, it also deserves to be mentioned here. When we deploy our bytecode into the network, it gets executed. Just executed, nothing more. In this execution, it's programmer's responsibility to setup the required environment then make the STOP instruction executed. This is necessary because otherwise EVM would try to execute also the functions. In Listing 1, you can see how we prepare our environment for EVM init phase.

Listing 1: Init Phase

```haskell
initPhase :: Integer -> CodegenM m => m ()
initPhase functionCosts = do
  rec op (PUSH32 functionCosts)
      op DUP1
      op (PUSH32 initCost)
      op (PUSH32 0x00)
      op CODECOPY
      op (PUSH32 0x00)
      op RETURN
      op STOP
      initCost <- use pc
  pure ()
```

As you might have noticed, we reference some future values in our code such as **initCost**. It may seem weird at first, but when we talk about laziness in section 5, we will describe how well Haskell handles these kinds of computations easily.

3.2 Code Phase

This phase is the heart of the EVM computation. In this part, we split bytecode into sections for each function. What each function does is similar to each other:

- Get the calling function name and test it to decide whether you are the one being called; if so, do nothing; else, jump to the next function test.
- Get the function parameters from the caller: functions can take multiple parameters from various types: Ivy handles only integer type parameters for now, but it handles multiple parameters, as much as caller wants.
- After testing and information gathering is finished, execute the function body.
- Execute the STOP instruction and return the result (s).

By looking only to the above sequence, one might assume that every function body should be executed in an isolated environment, which is not true in EVM's case because it's just a simple execution machine. From functions' body we can jump to whereever we want, in fact, this is how we manage internal function calls. Also one more important point is that the memory is shared: a smart contract is a stateful entity and it's memory state is persistent: function calls can actually mutate them. In fact, we plan to use this fact to manipulate distinguishment of function call from outer world from the internal function calls.

Our code for preparing function test and gathering environment information is a little bit low-level for now. This makes it hard to work on it, but we plan to abstract away most of the tedious low-level stack and memory management parts so that programmer won't have to think in terms of a primitive virtual machine, but in terms of abstract concepts.

In Listing 2, you can see a part of our code which tests and prepares a function to be executed.

Listing 2: Code Phase

```
codegenFunDef :: CodegenM m => FunStmt -> m ()
codegenFunDef (FunStmt sig@(FunSig _mods name args) block retTyAnnot) ↩
    = do
  sigToKeccak256 sig >>= \case
    Nothing -> throwError $ InternalError $ "Could not take the ↩
        keccak256 hash of the function name: " <> name
    Just fnNameHash -> do
      resetMemory
      registerFunction name args
      rec
          -- Function's case statement. If name does not match, we don↩
              't enter to this function.
          offset <- use funcOffset
          op (PUSH4 fnNameHash)
          op (PUSH1 0xe0)
          op (PUSH1 0x02)
          op EXP
          op (PUSH1 0x00)
          op CALLDATALOAD
          op DIV
          op EQ
          op ISZERO
          op (PUSH32 (functionOut - offset))
          op JUMPI

          -- Store parameters
          storeParameters args

          -- Function body
          funPc <- use pc
          Operand retTy retAddr <- executeFunBlock block
          checkTyEq "function definition" retTyAnnot retTy
          functionOut <- jumpdest
      updateCtx (M.insert name (TFun retTy, FunAddr funPc retAddr))
      return ()
```

4 Deployment

Deployment a smart contract in its essence is not directly related to EVM, but is a necessary procedure if we are developing a programming language for writing smart contracts.
There are several useful scripts we use in order to perform a deployment. This is manually done, but we also have an in-progress automated deployment and testing environment being developed in Nix, which we will talk about in details in section 7.

4.1 Ivy.sh: Shell Scripts

While testing Ivy's features, we wrote several shell scripts for convenience and eliminating duplicated work and combined it in the file *ivy.sh*: we now realize that this is actually our compiler script. It has the following functionalities:

- compile: Compile the Ivy compiler
- ast: Run the ivy compiler on file *stdlib.ivy* and print only the abstract syntax tree for the syntax in *stdlib.ivy*.
- bytecode: Run the ivy compiler on file *stdlib.ivy* and print only the bytecode produced from it.
- run: Run the ivy compiler on file *stdlib.ivy* and also execute it in EVM and print out the results to stdout.
- disasm: Run the ivy compiler on file *stdlib.ivy* and print out the EVM assembly it produced. This is helpful to track down and debug instructions, especially JUMP instructions.
- deploy: Perform a deployment using the deployment script *deployment/deployment.js*. We mutate this script to insert bytecode, so we create an old version postfixed with *.backup*.
- rewind-deploy: Restore *deployment/deployment.js* with *deployment/deployment.js.backup*.
- compute: Compile the compiler, run it and if it succeeded, deploy it else print a descriptive error message. This is the most frequently used script while testing Ivy's functionalities.

4.2 Procedure

In order to deploy a bytecode, we need a way to communicate with an existing blockchain network. We use testrpc[4] for starting a test network to which we can deploy our smart contracts and in which we can run them. After we have a running network, we are ready to deploy our smart contract which is just a long bytecode.

Our deployment script is actually a nodejs[5] program which has dependencies and the main script. We make use of web3.js[6] for communicating with the network. Web3.js provides a nice and high-level api for communicating with the network and smart contracts.

We won't go in details of the script since it's pretty self-explanatory in terms of its behavior, but as a high-level explanation, what it does is to connect a local port in our machine, select the first account available on the network and send a transaction from this account with including the bytecode to use.

5 Laziness

Laziness is an evaluation strategy that is employed by several programming languages, some of them enable it by default and make strict evaluation an option (eg. Haskell[7]), and some of them disable it by default and leave it as an option to the programmer to use lazy evaluation (eg. Scala[8]).

Lazy evaluation is basically evaluating expressions when they are needed, not when they are declared. For instance, in Haskell, an expression like $let\ v = 3+5$ is not actually computed until we tell compiler that we want the value of v. The "value" part is important because it's not enough to tell we want v, but we want it's "value" and also using the fact that it's a integer. What happens if we pass v to a function as parameter, then? It's passed as a *thunk*. A *thunk* is basically a value that compiler knows how to compute, but is not computed yet. This simple idea lets us come up with various optimizations in Ivy, and they are illustrated in their own sections.

By using thunks, we are able to simulate an infinite memory (even if it is not, we don't need to know its limits) (see section 5.1), we are able to jump to places of which we don't know the program counter yet (see section 5.2).

5.1 Infinite Memory Model

As we described before, EVM has a rather primitive memory representation and we need to model our compiler in such a way that it allows allocating new variables. While we are looking for a place to allocate space for our variable, we need to traverse memory from where we left, and look for an available empty space.

5.1.1 Explanation

In Haskell, [*value*..] expression represents a list from *value* to infinity. The reason our program does not explode just when we use this expression is that lists are also lazy evaluated so that we can pick values one by one from it until we find the value we are interested in. We mentioned that we keep a pointer to denote the location we've left while allocating for variables, and this pointer is our starting value for searching for an empty space.

Lazily searching through memory locations makes it easy to allocate array spaces as well. Say we want to allocate a n-length array, and we need n consecutive empty memory spaces for array allocation. We can search for these spaces easily by lazily testing memory cells.

5.1.2 Concerns and Limits

Modeling a memory as if it's infinite is of course not something we would use for production, we need a way of saying "this operation costs more memory than it should", but for now, we won't have such smart contracts so we can skip this limitation for the time being.

5.2 Lazy Jumping

Lazy jumping is the most mind-blowing implementation detail of Ivy, it's also a good way of demonstrating how to model computations which need some future values. We already showed an example of lazy jumping, but now we present more sophisticated examples from the Ivy source code:

Listing 3: Lazily Jumped While Loop

```haskell
rec op (PUSH32 (whileOut - offset))
    op JUMPI

    -- Loop start
    loopStart <- jumpdest

    -- Prepare true value of current PC
    op (PUSH32 (loopStart - offset))

    -- Code body
    executeBlock block

    -- Load predicate again
    Operand predTy' predAddr' <- codegenExpr pred
    checkTyEq "index\_of\_while\_pred" TBool predTy'
    load predAddr'
    -- op SWAP1

    -- Jump to destination back if target value is nonzero
    op SWAP1
    op JUMPI
    whileOut <- jumpdest
pure ()
```

Listing 4: Lazily Jumped If Statement

```haskell
  Operand tyPred addrPred <- codegenExpr ePred
  checkTyEq "if\_expr" tyPred TBool

  load addrPred
  op ISZERO -- Negate for jumping condition

 -- offset <- estimateOffset bodyBlock
  rec op (PUSH32 ifOut) -- +3 because of the following `PC`, `ADD` and↩
      `JUMPI` instructions.)
      -- op PC
      -- op ADD
      op JUMPI

      void $ executeBlock bodyBlock
      ifOut <- jumpdest
  pure ()
```

The task of jumping through instructions is actually not a problem with assembly languages with labeling, but EVM is also primitive in this sense and does not support it so we need to know the PC of every place we are jumping to. This is the primary reason to use lazy jumping.

Lazy evaluation with such usage does not come for free, it has it's own dangerous points. As you can see, we use a function *op*() everywhere we want to write an instruction. This function actually does not compute the parameter, which is the instruction, but instead it adds the instruction to a accumulating list to be unfolded later. In other words, we create our program which consists of several instructions, and at the end we unfold them into a bytecode. This procedure is required because if we were forcing instructions to be evaluated right away, the dependencies would form an infinite recursion. This is an important aspect of using such computation model.

6 Testing

In this term, we introduced automated tests for Ivy. We have three source codes for now, each testing different aspects of Ivy sources. A brief summary of tests that currently exist:

1. Testing parser with a source file with corrupted syntax, which should fail
2. Testing if branching by different value assignments in each branch then asserting the right one
3. Testing computation of 15th fibonacci series element and asserting that value should exist in memory afterwards
4. Testing computation of 15th fibonacci series element for false case and asserting value to not exist in memory

Tests make our development flow smoother since we are now able to test whether we broke some part of the compiler or not, by just running tests.

We also perform more sophisticated tests with Nix, which will be described in its own section.

7 Automated Testing of Smart Contracts Using Nix

Nix[9] is a purely functional package manager, and NixOS[10] is a purely functional linux distribution. We make use of Nix for automated testing.

Here is the task of testing a smart contract compiled with Ivy:

1. Compile Ivy, compile the Ivy source file and extract the bytecode it generates
2. In a separate machine, run a blockchain test network on a local port
3. Connect to blockchain network we created in step 2 and run the deployment script with the bytecode we generated in step 1

4. Get the returned value from the blockchain network and assert whether that value is the expected one, and report a descriptive error message.

These kind of tests where we need several machines with different systemd services (which is used to keep processes alive in unix-like operating systems) involves so much complexity but with a declarative language, it's easy to overcome and this is why we decided on using Nix.

With Nix, we first describe two machines, one for blockchain the other one for our client, or programmer who is using Ivy to write smart contracts. A blockchain network wakes up, notifies the client and then client runs the deployment script to connect to blockchain and deploy the contract bytecode.

Implementation is in progress and can be found in the directory $nix-tests/$.

8 Conclusion

This term was all about improvement and enhancements to the current development environment of Ivy, and also some improvements and simplifications in memory-side. We had opportunities to try to combine different technologies such as web3.js, testrpc and Nix in order to make it easy to test and deploy Ivy programs.

References

[1] CMPE491 Midterm Progress Report:
https://github.com/yigitozkavci/ivy/blob/master/standard/report_491_midterm/report.pdf

[2] CMPE491 Final Progress Report:
https://github.com/yigitozkavci/ivy/blob/master/standard/report_491_final/report.pdf

[3] Yellowpaper: http://yellowpaper.io/

[4] Testrpc: https://github.com/trufflesuite/ganache-cli

[5] Node.js: https://nodejs.org/en/

[6] web3.js: https://github.com/ethereum/web3.js/

[7] Haskell Programming Language: https://www.haskell.org/

[8] Scala Programming Language: https://www.scala-lang.org/

[9] Nix: The Purely Functional Package Manager: https://nixos.org/nix/

[10] NixOS: The Purely Functional Linux Distribution: https://nixos.org/