Cmpe491 Midterm Progress Report

Yiğit Özkavcı

October 2017

Contents

1 Introduction **2**

2 Language Constraints by EVM **2**
2.1 Gas Cost 2
2.2 Entrance Point 3
2.2.1 Construction 3
2.2.2 Code to execute upon receiving a message 3
2.3 Message Sending 3

3 Optimisations **4**
3.1 Stack 4
3.2 Memory 4

4 Implementation **4**
4.1 Lexing 4
4.2 Parsing 4
4.3 Type Checking 4
4.4 Code Generation 4
4.5 Optimisation Passes 4

1 Introduction

This is the midterm report for project Ivy, a programming language for writing smart contracts on Ethereum Virtual Machine.

Ethereum is a decentralized plaform that runs smart contracts: pieces of codes that have the ability to run on any blockchain network. In order to write smart contracts, one should deploy a EVM-executable bytecode into blockchain network, which is not practical since bytecode is a sequence of hex characters; nothing more. To overcome this problem, EVM-compatible programming languages are being designed in order to abstract the problem of having to deploy plain bytecode.

There are already programming languages targeting EVM, including Solidity[1], Bamboo[2] and Viper[3]; and Ivy is planning to be a programming language that is easy to use, ability to scale with abstractions and capable of generating an optimised bytecode.

2 Language Constraints by EVM

Designing a programming language while targeting EVM has several problems that general purpose programming languages don't. In this section, we will investigate problems that we have / may encounter.

2.1 Gas Cost

Basically, in order to run compurations on a blockchain via smart contracts, one should pay **enough or more** gas. But... what is gas? Gas is an alias for Wei, the smallest unit of subdenomination of Ether. Below are differend kinds of subdenominations and their multiplier of Wei.

Wei Multiplier	**Name**
10^0	Wei
10^12	Szabo
10^15	Finney
10^18	Ether

Table 1: Ether Subdenominations

Gas is the pricing of computations that are run by smart contracts. Basically, each interaction via a smart contract must be paid in units of gas. This also means that the higher abstraction level we have for EVM computations, the more costly our computations will be, because of the abstraction layer switch of the compiler.

This is where compiler optimizations are critically important. In the context of smart contracts, compiler optimization is not just about how much of the RAM or CPU of the user you consume, but also the real money of the user you waste in the runtime. We will investigate optimisations in terms of stack operations (see 3.1) and memory allocations (see 3.2) in further sections.

2.2 Entrance Point

A smart contract has two main phases in its life time: construction and listening. We will illustrate both of them in the following sections, and also explain how to adapt these two distinch phases into Ivy's strategy of logic flow.

2.2.1 Construction

When a smart contract is deployed on EVM, a code is being executed in the same fashion Java[4] constructs instances of objects described in their class. In this phase, an EVM-oriented programming language should give programmer the address information of the deployer, and let user do whatever they want with it. Typically, users set that address information in object-oriented EVM-targeting languages like Solidity[1]

This phase is one shot: meaning construction phase will be terminated once it finishes its execution. It's programmer's responsibility to use this initial phase according to the program's needs.

2.2.2 Code to execute upon receiving a message

The more interesting phase is the second phase. As we said before, a smart contract is not like a terminating program, but an executable (see 2.3) with internal state (see 3.1 and 3.2). Hence, a program written in a language shall implement a parametric and callable interface which can be run from outer world.

Not every call to a contract shall necessarily change the internal state of it. There are mainly two kinds of messages: reading data from the state and updating it. Reading data doesn't cost any gas, but updating the state surely does.

2.3 Message Sending

If one wants to read or update data on a smart contract, they should execute a procedure on that contract. As we noted before, contract should describe this executable code interface on construction phase (see 2.2.1). Say we have the following interface for our contract:

```
/*
 * An abstract address representation
 */
abstract class Address {
  public int value;
}

/*
 * Contract interface that will be used from outer world
 */
interface ExampleContract {
  /*
   * Returns how many cats account in with given address adopted
   */
  public int getAdoptedCatCount(Address address);

  /*
   * Adopt the cat with given id by sender address
   */
  public void adopt(int catId);
}
```

3 Optimisations

3.1 Stack

3.2 Memory

4 Implementation

4.1 Lexing

4.2 Parsing

4.3 Type Checking

4.4 Code Generation

4.5 Optimisation Passes

References

[1] Solidity: Contract-Oriented Programming Language, https://github.com/ethereum/solidity

[2] Bamboo: a morphing smart contract language, https://github.com/pirapira/bamboo

[3] Viper: an experimental programming language targeting EVM, https://github.com/ethereum/viper

[4] https://en.wikipedia.org/wiki/Java_(programming_language)