Cmpe491 Midterm Progress Report

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

October 2017

Contents

1 Introduction

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

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 computations 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 4.5) and memory allocations (see 4.6) 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 4.5 and 4.6). 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:

```java
// code/ContractInterface.java

/*
 * 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 Language Description

3.1 Abstract Syntax Tree

```haskell
-- code/Syntax.hs

data Op =
    OpAdd
  | OpMul
  | OpSub
  | OpDiv
  deriving Show

type Length = Integer
type Index = Integer

data PrimType =
    IntT
  | CharT
  | Array Length PrimType
  deriving (Eq, Show)

data Expr =
    IntExpr Integer
  | CharExpr Char
  | Identifier Name
  | VarDecl PrimType Name
  | Assignment Name Expr
  | ArrAssignment Name Index Expr
  | BinaryOp Op Expr Expr
  | Times Integer Block
  | Debug Expr
  deriving Show
```

3.2 ABNF Syntax

```
; ABNF Syntax for ivy, a programming language for ethereum virtual ↩
    machine
; based on RFC 5234 (https://tools.ietf.org/html/rfc5234).

equal          = "="  whitespace
plus           = "+"  whitespace
minus          = "-"  whitespace
times          = "*"  whitespace
div            = "/"  whitespace
line-comment   = "--" whitespace
single-quote   = "'"  whitespace
opening-bracket = "["
closing-bracket = "]"

operator =
    plus
  / minus
  / times
  / div

tab = %x09 ; \t

ALPHA =
    %x41-5A ; [A-Z]
  / %x61-7A ; [a-z]

DIGIT = %x30-39 ; ASCII 0-9

ALPHANUM = *(ALPHA / DIGIT)


single-whitespace =
    " "
  / tab
whitespace = *single-whitespace

intType = %x69.6E.74 ; int

charType = %x63.68.61.72 ; char

primType = intType / charType

arr-type = primType opening-bracket integer closing-bracket

type =
    primType
  / arr-type

timesBegin = %x74.69.6D.65.73 ; times
timesEnd = %x65.6E.64 timesBegin ; endtimes

integer       = *DIGIT
char          = single-quote ALPHANUM single-quote
identifier    = *ALPHA
var-decl      = type identifier whitespace equal whitespace expr
assignment    = identifier whitespace equal
arrAssignment = identifier opening-bracket integer closing-bracket ↩
    whitespace equal expr
binop         = expr whitespace operator expr
timesExpr     = integer "." timesBegin expr timesEnd

expr =
    integer
  / char
```

```
  / identifier
  / var-decl
  / assignment
  / arrAssignment
  / binop
  / timesExpr
```

4 Language Implementation

This section contains a complete information about the implementation of Ivy from lexing to all the way to code generation & planned optimisations. The compiler of the Ivy language is being implemented in Haskell. Also for lexing, parsing and code generation, no other software is/planning to be used.

4.1 Lexing

Ivy lexer is actually does so little right now: it:

- Defines operators (eg. +, *, -, ;, ...)
- Defines reserved names (eg. if, else, times, endtimes, ...)
- Eliminates commented lines (starting with $--$)
- Defines special token types that are helpful for parser (see 4.2) (eg. integer, charLiteral, parens, ...)

4.2 Parsing

Ivy parser is responsible of taking a string and generating the AST of the Ivy language (see 3.1).

We are using monadic parser combinators inside Parsec library. Ivy parser is a top-down recursive-descent parser with backtracking ability, meaning it can return and try different inputs in the case of an error. In fact, this kind of parsing and error aggregation nicely plays along with monadic parsing.

Advantages of using a monadic parsing hence not using a applicative parsing is beyond this report's scope, but it's indeed a critical decision to make when it comes to using one through parsing of whole compile inputs.

The top-level of our parser is illustrated below:

```
-- code/Parser.hs

factor :: Parser Expr
factor = try (parens expr <?> "parens")
     <|> try timesIterationBegin
     <|> try prims
     <|> try varDecl
     <|> try arrAssignment
     <|> try assignment
     <|> try identifier
     <|> debug
     <?> "factor"
```

4.3 Code Generation

Code generation in Ivy language differs from general purpose programming languages in the sense that it **should** target the EVM platform by creating a big chunk of bytecode. This disallows us from utilising tools like LLVM[5] on code generation and optimisation phases.

Code generation module is Ivy's most complex module at the time of writing. It's responsible of receiving an AST (see 3.1) generated by parser (see 4.2) and create a bytecode that will be consumed by EVM.

Ivy's codebase includes a backend for communicating with the low-level EVM code. This EVM API doesn't contain any abstraction (except memory operations; see 4.6 for details), only a type-safe interface for EVM instructions and the code generator makes use of this instraction in its logic flow. Following is an example of usage of EvmAPI module from one of codegen module's consumption:

```
-- code/Codegen.hs

codegenTop :: Expr -> Evm (Maybe Operand)
codegenTop (Times until block) = do
  -- Assign target value
  op2 PUSH32 until
  op JUMPDEST

  -- Prepare true value of current PC
  op PC
  op2 PUSH32 0x01
  op SWAP1
  op SUB

  -- Decrease target value
  op SWAP1
  op2 PUSH32 0x01
  op SWAP1
  op SUB

  -- Code body
  executeBlock block
  -- Jump to destination back if target value is nonzero
  op DUP1
  op SWAP2
  op JUMPI

  return Nothing
```

4.4 Execution

In order to execute code, Ivy will be eventually tried in *ebloc* project of the Boğaziçi University. Now, for debug purposes, we make use of Go[6] implementation of the ethereum virtual machine[7].

4.5 Stack

EVM has two components for storing data: stack for temporary data (like taking summation of two numbers), and memory for the rest. Simply, if one wants to store any information that can be used in the future, they should store it in the memory.

In order to manipulate stack, we have PUSH and POP instructions. It's using an assembly-like stack model but has less in it in the sense of instructions and ability to manipulate different registers along the way.

Instructions like ADD, DIV and JUMP takes 'parameter's from the stack, and push zero or more items into it. All of these instructions are well defined in the yellowpaper[8].

Especially jumping mechanism in EVM has right to be mentioned. Execution flow in EVM doesn't have the notion of line numbers, but instead, it has markers. This makes jumping a lot harder than the traditional assembly language implementations. One should first declare where is a legit jumping point, do their computations and jump to that specific place.

4.6 Memory

EVM's memory model is pretty simple: it has cells of 32 bytes, and most of the instructions (except MLOAD) give us options to work with 1 to 32 bytes depending on our needs.

We wanted users declare primitives they need with the byte size they want (like uint8_t, uint32_t, ... in C programming language); hence we needed to come up with an efficient memory algorithm that could allow filling up cells without leaks occuring. Following is the source code of the algorithm:

```
-- code/MemAlgorithm.hs

alloc :: Size -> Evm Integer
alloc size = do
  memPtrs <- use memPointers
  case M.lookup size memPtrs of
    Nothing -> throwError $ InternalError $ "Pointer does not exist: "↩
        <> show size
    Just (MemBlock index alloc) ->
      if totalMemBlockSize - alloc >= sizeInt size
        then
        let
          newPos :: Integer = (alloc + sizeInt size)
        in do
          updateMemPointer size index newPos
          let baseAddr = calcAddr index alloc
          let targetAddr = calcAddr index newPos
          markMemAlloc index targetAddr
          return baseAddr
      else do
          newIndex <- findMemspace
          let baseAddr = 0
          let targetAddr = sizeInt size
          updateMemPointer size newIndex targetAddr
          markMemAlloc newIndex targetAddr
          return (calcAddr newIndex baseAddr)

allocBulk
  :: Integer
  -> Size
  -> Evm Integer
allocBulk length size = do
  mem <- use memory
  let msize = fromIntegral $ M.size mem
  if sizeInt size * length <= totalMemBlockSize
     then -- There are 5 blocks of 4 bytes
       memory %= M.update (updateInc size length) msize
     else do -- There are 15 blocks of 4 bytes
       let fitinLength = totalMemBlockSize `div` sizeInt size -- 32 / ↩
           4 = 8 mem blocks can fit in
       memory %= M.update (updateInc size fitinLength) msize
       void $ allocBulk (length - fitinLength) size
  return $ calcAddr msize (0 :: Integer)
    where
      updateInc :: Size -> Integer -> Integer -> Maybe Integer
      updateInc _ 0 allocated = Just allocated
      updateInc size length allocated = updateInc size (length - 1) (↩
          allocated + sizeInt size)
```

4.7 Type Checking (Planned)

4.8 Optimisation Passes (Planned)

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)

[5] LLVM: The compiler infrastructure project: https://llvm.org/

[6] Go programming language: https://golang.org/

[7] Official Go implementation of the Ethereum protocol: https://geth.ethereum.org

[8] Ethereum: a secure decentralised generalised transaction ledger: http://gavwood.com/paper.pdf