Cmpe491 Final Progress Report

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

October 2017

Contents

1 Introduction

This is the CMPE491 final 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 5.5) and memory allocations (see 5.6) 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 5.5 and 5.6). 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:

```
// 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

```
-- code/Syntax.hs

data Op =
    OpAdd
  | OpMul
  | OpSub
  | OpDiv
  | OpMod
  | OpGt
  | OpLt
  | OpEq
  deriving (Eq, Show)

type Length = Integer
type Index = Integer

data PrimType =
    TInt
  | TChar
  | TBool
  | TArray Length PrimType
  | TFun PrimType
  deriving Eq
```

```
data Stmt =
    SVarDecl PrimType Name
  | SDeclAndAssignment PrimType Name Expr
  | SAssignment Name Expr
  | SArrAssignment Name Expr Expr
  | STimes Integer Block
  | SWhile Expr Block
  | SIf Expr Block
  | SIfThenElse Expr Block Block
  | SReturn Expr
  | SExpr Expr
  deriving (Eq, Show)

data SFunDef = SFunDef String [(PrimType, Name)] Block PrimType
  deriving (Eq, Show)

data AnyStmt =
    FundefStmt SFunDef
  | Stmt Stmt
  deriving (Eq, Show)

data Expr =
    EInt Integer
  | EChar Char
  | EBool Bool
  | EIdentifier Name
  | EArrIdentifier Name Expr
  | EBinop Op Expr Expr
  | EFunCall String [Expr]
  | EArray Length [Expr]
  deriving (Eq, Show)
```

3.2 ABNF Syntax

We formally defined Ivy's abnf syntax based on the specification given in RFC5234[9]

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

; Arithmetic operators
plus           = "+"   whitespace
minus          = "-"   whitespace
mul            = "*"   whitespace
div            = "/"   whitespace
mod            = "%"   whitespace

; Comparison operators
greater-than   = ">"   whitespace
less-than      = "<"   whitespace
equals         = "=="  whitespace

; Other operators
equal          = "="    whitespace
open-bracket   = "["   whitespace
close-bracket  = "]"   whitespace
open-parens    = "("   whitespace
close-parens   = ")"   whitespace
stmt-separator = ";"   whitespace
stmt-end       = stmt-separator whitespace end-of-line
line-comment   = "--"  whitespace
```

```
single-quote   = "'"  whitespace
dot            = "."
curly-open     = "{" whitespace
curly-close    = "}" whitespace
comma          = "," whitespace

; Hex characters for case senstivity
; These char sequences can be generated with the following Haskell ↩
    function:
;
; convert :: String -> String
; convert = ("%x" ++) . intercalate "." . map (flip showHex "" . ord)
int      = %x69.6E.74
char     = %x63.68.61.72
bool     = %x62.6F.6F.6C
while    = %x77.68.69.6C.65
times    = %x74.69.6D.65.73
endtimes = %x65.6E.64 times
if       = %x69.66
return   = %x72.65.74.75.72.6e
else     = %x65.6c.73.65
true     = %x74.72.75.65
false    = %x66.61.6c.73.65

; Utilities
end-of-line =
    %x0A     ; "\n"
  / %x0D.0A  ; "\r\n"

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

DIGIT = %x30-39 ; ASCII 0-9

ALPHANUM = *(ALPHA / DIGIT)

tab = %x09

single-whitespace =
    " "
  / tab

whitespace = *single-whitespace

operator =
    plus
  / minus
  / mul
  / div
  / mod
  / greater-than
  / less-than
  / equals

prim-type =
    int
  / char
  / bool

arr-type = prim-type open-bracket int-literal close-bracket

type =
    prim-type
  / arr-type

; Statements
block              = curly-open end-of-line stmts curly-close ↩
```

```
    whitespace stmt-end
var-decl            = type identifier whitespace equal whitespace expr
assignment          = identifier whitespace equal
typed-assignment    = prim-type assignment
decl-and-assignment = var-decl whitespace equal whitespace expr
arr-assignment      = identifier open-bracket int-literal close-↩
    bracket whitespace equal expr
binop               = expr whitespace operator expr
times-stmt          = int-literal dot times block endtimes
while-stmt          = while whitespace open-parens expr close-parens
if-stmt             = if open-parens expr close-parens block
if-then-else-stmt   = if-stmt else block

stmt =
    var-decl
  / decl-and-assignment
  / assignment
  / arr-assignment
  / times-stmt
  / while-stmt
  / if-stmt
  / if-then-else-stmt
  / return
  / expr

stmts =
    stmt whitespace stmt-end
  / stmt whitespace stmt-end stmts

; Expressions
int-literal       = *DIGIT
char-literal      = single-quote ALPHANUM single-quote
bool-literal      = true / false
identifier        = *ALPHA
arr-identifier    = identifier open-bracket expr close-bracket
comma-sep-expr = expr / expr comma comma-sep-expr
func-call         = identifier open-parens comma-sep-expr close-parens
array-expr        = curly-open comma-sep-expr curly-close

expr =
    int-literal
  / char-literal
  / bool-literal
  / identifier
  / arr-identifier
  / binop
  / func-call
  / array-expr
```

4 Language Features

4.1 Branching

Branching directly corresponds to if-else statements in Ivy language. We currently have two statements, which we plan to convert to expressions: if and $if - then - else$. It's already intuitive to figure out what they do, so we'll only talk about their implementation in EVM's memory.

Branching in a virtual machine with primitive instructions is pretty straightforward. You have JUMP and JUMPI(conditional jump) instructions, and execute them based on the top level value in the stack.

The plan: if the predicate inside if's condition evaluates to true, we do nothing; that is we enter to the if block. Else, we should skip this body, but how can we know how much PC we should skip with this jump?

In this task, we had to introduce capturing PC, the program counter in order to estimate which PC we should jump to, for skipping certain statement bodies. After we started capturing PC, another challenge came up: how do we know how much PC do a bunch of instructions cost? We had to also give compiler the information of how much PC each instruction takes, and it turns out that some instructions such as PUSH32 costs more (33) PC than an arbitrary other instruction such as JUMP (1). With this in mind, we were able to see the future, and jump into that future.

4.2 Looping

Looping mechanism is very similar to branching, only difference is being we need to evaluate the predicate again an again, and load it into the same memory cell in order to memorize it inside the block. For instance, a predicate $(a > 3)$ should be re-evaluated in every execution of the loop because we might be modifying the variable a in the process.

A problematic functionality is being introduced into Ivy here: infinite calls. EVM is smart enough not to run out of memory in these cases, but we should be handling these cases in the language itself. For now, we cannot do anything in these cases but this is a problem to be aware of.

With the looping added along with the branching, we now have a Turing-complete language! We can basically compute any well-defined problems with Ivy.

4.3 Functions

Function-calls were real challenge because of the way lexical scope & memory work. We needed to figure out what we do in compile-time and what's being done in run-time extensively. Here is the implementation of defining a function:

We are well-aware of the fact that while we are defining a function, EVM should not execute it, but skip it. Because of this, function definition in instruction-side involves a jump statement at the beginning of the function body. This way, we define our function but jump around it, and place a $JUMPDEST$ inside it for further calls.

To call a function, we needed to think of both our lexical scope, and the runtime modifications that will be made by the function. On calling a function, we prepare our parameters (this will be discussed in the following paragraph), put our address into stack for returning back and jump to the function body destination. After this, we execute the function body in EVM and return back to the position on the stack & pop it.

Preparing function parameters are weird, because in nature, you should have runtime values for them, but you should also make use of them in function definition's instructions. In order to achieve this, we pre-processed functions and stored their parameters in defined cells in memory. This way, we know where to load parameters from in definition, as well as where to store them when calling.

We mentioned a pre-processing for functions above. This was a decision invoked by function parameters but it also helped Ivy being able to call functions before defining them. With this in mind, we already registered function signatures to lexical scope while we register their parameters.

5 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.

5.1 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 5.2) (eg. integer, charLiteral, parens, ...)

5.2 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:

```haskell
-- code/Parser.hs

stmt :: Parser Stmt
stmt =
  try declAndAssignment
   <|> try times
   <|> try while
   <|> try varDecl
   <|> try arrAssignment
   <|> try assignment
   <|> try sIfThenElse
   <|> try sReturn
   <|> try sExpr
   <|> sIf
   <?> "Statement"

expr :: Parser Expr
expr = buildExpressionParser binops expr'
  where
    expr' :: Parser Expr
    expr' = try (parens expr <?> "parens")
         <|> try ePrim
         <|> try eFunCall
         <|> try eArrIdentifier
         <|> eIdentifier
         <?>  "factor"
```

5.3 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 5.2) 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 5.6 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
```

5.4 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].

5.5 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 lan-

guage implementations. One should first declare where is a legit jumping point, do their computations and jump to that specific place.

5.6 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. Here is the description for algorithm:

We allocate memory cells with sizes of 1-byte, 2-bytes, 4-bytes, 8-bytes and 32-bytes. In order to achieve maximum efficiency here, we keep pointers for each type of size. For instance, if 2-bytes has a pointer to cell 84, next allocation for a 2-bytes size data will be placed on address 84 and pointer will find another place for itself. This location may not necessarily have the address 86, algorithm decides another suitable position for this allocation.

5.7 Type Checking

Ivy compiler's typechecking is done in Codegen module, which manages the code generation phase. An external type-checker is planned, but for now, we type-check as we generate the code.

Type checking phase has its own error types, summarized below:

- TypeMismatch
- ScopedTypeViolation
- WrongOperandTypes
- ArrayElementsTypeMismatch

In the lookup table, Ivy compiler stores variable addresses as well as their types. This means that all the types are, and should be available to the compiler; this does not prevent compiler from inferring some types though, type inference will be discussed in 5.8 further. With this in mind, whenever we see an operand, and it was declared, we already know its type and hence we can type-check it.

For arrays, types are defined exactly like primitive types, but involves a more complex type-checking mechanism since we have array-value expressions like $1, 3, 5$. This brings up the question of array element types not being the same, and we do have a process for it. We immediately throw the error $ArrayElementsTypeMismatch$ in a case of mismatched type for elements.

I've personally found http://www.cse.chalmers.se/edu/year/2015/course/DAT150/lectures/proglang-07.html lecture **very** useful for implementing typing judgements for a type-checker module. Since we can already figure out types for values and expressions,

implementing the judgements will only be putting another layer of constraint to the compiler.

5.8 Type Inference (Planned)

We plan to initially implement type inference with a *auto* keyword used instead of a type annotation, inspired by $C++$ language. This inference will initially be trivial to implement since we already know the types exactly. A more complex type inference will be introduced after we adopt the functional paradigm in Ivy.

5.9 Optimisation Passes (Planned)

We plan to implement LLVM-like optimisation passes, which are going to touch and optimise specific parts of the program. They can act on both before the compilation or on the bytecode itself while optimising the generated bytecode.

6 Future Work

We aim Ivy to be a pure-functional language with effects of smart contract being able to described at type-level in function signatures. This behavior can be achieved via effect-handling model of Elm or Purescript programming languages. With this in mind, possibilities are very rich and make important progress in field since writing safe smart contracts are crucial right now.

Changing the paradigm of a language entirely is often thought to be impossible / too hard, but since we already splitted the expression and statements from each other, we only need to provide purity and immutability through the syntax and the actions of the compiler.

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

[9] Augmented BNF for Syntax Specifications: ABNF (RFC-5234) https://tools.ietf.org/html/rfc5234