Ivy: A programming language targeting EVM

Yigit Ozkavci
Advisor: Can Ozturan

Introduction

Ivy is a programming language designed for Ethereum Virtual Machine. Ivy enables us to write smart contracts and deploy them directly to an Ethereum blockchain.

Motivation

Smart contracts are widely accepted concept of transferring values without the existence of a middleman. A smart contract lives in the blockchain like any other entity, and is able to receive & send valuables, with the decision being made via contract's internal state.

Every computation on the smart contracts costs gas, a measure for computational power required by the contract; this makes the content of smart contract even more critical since the efficiency of computation directly affects the costs of executing the contract, in real money.

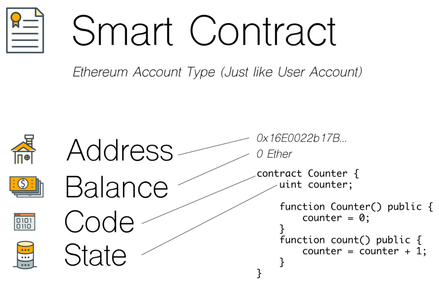

Figure 2: Smart Contract with Solidity

Language Features

Ivy is a Turing-complete programming language, meaning given enough memory, it can run any given computation.

Below is a list of features that Ivy currently supports:

- Primitives: int, char, bool
- Array types: int[], char[], bool[]
- Branching (if-else statements)
- Loops
- Functions
- Arithmetic (+,-,*,/) and comparison (>,==,<) operators
- Type-checking
- Parsing error handling
- Compile-time error handling:
  - VariableNotDeclared
  - VariableAlreadyDeclared
  - VariableNotDefined
  - TypeMismatch
  - ScopedTypeViolation
  - WrongOperandTypes
  - FuncArgLengthMismatch
  - ArrayElementsTypeMismatch
  - NoReturnStatement

Architecture

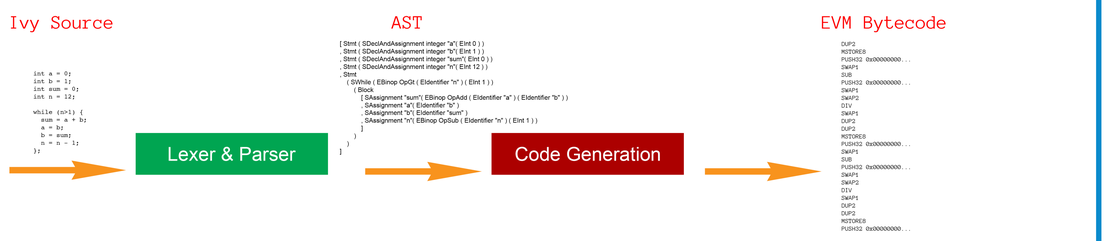

Figure 1: Ivy Architecture

Ivy compiler first runs the lexer to tokenize ivy source into meaningful pieces. Then parser works through these tokens and creates an abstract syntax tree.

This abstract syntax tree directly corresponds to a top level recursive Haskell type called $AnyStmt$. As you might guess, this denotes that an Ivy program is just a big statement, consisting of smaller statements ($Stmt$ type) and expressions ($Expr$ type).

You might wonder why we don't have a top level $Stmt$ type, but $AnyStmt$; this is because in Ivy, we pre-process function definitions so that we can call them before their declaration in our program. Hence, $AnyStmt$ expands into $FundefStmt$ and $Stmt$.

The magic happens in code generation phase. $Codegen$ module takes ast as input and generates EVM bytecode.

Research

Before writing Ivy's compiler, we first needed to explore how EVM works, and how it manages stack & memory to persist temporary computation steps.

There is one and only one source that describes how EVM works, which is called the **yellowpaper**[1] which is a very-formally designed paper that explains every operation that could be executed on EVM in detail.

After we figured out how memory is handled, the rest was to learn instruction set of EVM, which itself has an assembly-like instruction set that runs instructions on stack. Compiling Ivy essentially the process of converting ivy source code into EVM bytecode with EVM bytecode being series of bytes, with each one representing an instruction to be executed.

References

[1] DR. Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger.

[2] Elm programming language: http://elmlang.org/.

[3] Purescript programming language: http://www.purescript.org/.

Future Research

Even though Ivy is currently usable, it still lacks several features such as external type-checker module, recursions, module system and a stable standard library.

We plan Ivy to be a pure functional language, hence we need to implement message sending & receiving in this paradigm. There are good examples such as Elm[2] and PureScript[3] which are pure functional front-end languages managing effects user-inputs in different styles.

Contact Information

Web https://github.com/yigitozkavci
Email yigitozkavci8@gmail.com
Phone +90 535 084 76 02