Smart Contracts Environments: similarities and differences

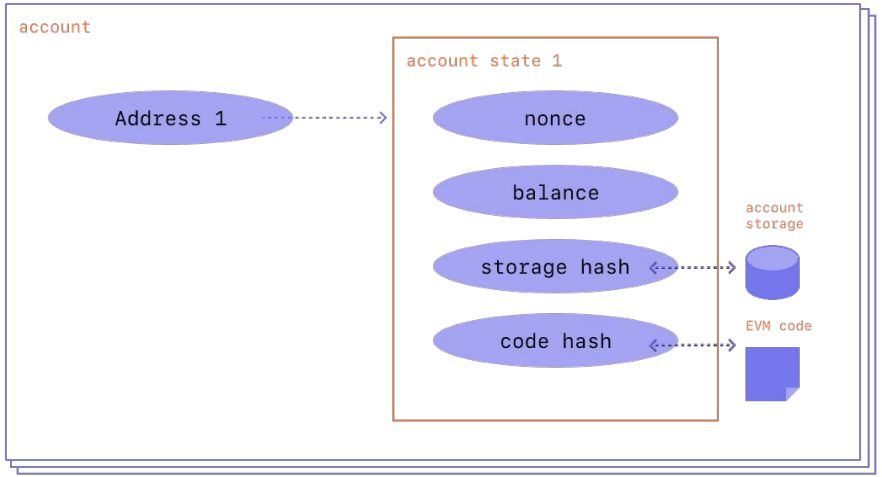
Lorenzo Benetollo

Introduction

Ethereum

Ethereum Smart Contracts

- Code that can be executed by the **EVM**
- Written in **Solidity** or other Languages (Vyper, Yul, Fe,)
- Account type with **Code** and **Storage**
- Uniquely identified by an **address**



Transactions

A transaction is a message that is sent from one account to another account, it can include binary data, which is called “payload”, and Ether.

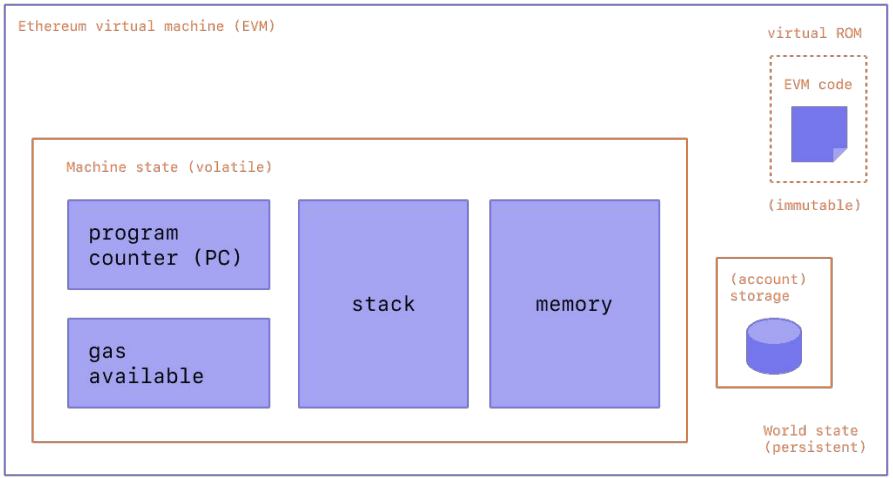
Transactions contains information organized in fields. Depending on what fields are specified, there can be three types:

- Regular transactions
- Contract deployment transactions (to Zero address and Data)
- Execution of a contract (data as input)

- `from` – the address of the sender, that will be signing the transaction
- `recipient` – the receiving address
- `signature` – the identifier of the sender generated by the sender's private key
- `nonce` - a sequentially incrementing counter which indicates the transaction number from the account
- `value` – amount of ETH to transfer
- `data` – optional field to include arbitrary data
- `gasLimit` – the maximum amount of gas that can be consumed by the transaction
- `maxPriorityFeePerGas` - the maximum price of the consumed gas as a tip to the validator
- `maxFeePerGas` - the maximum fee per unit of gas willing to be paid

Ethereum Virtual Machine

- Executes as stack stack machine
- Runs machine code obtained by source code compilation
- Alters Ethereum state by performing state transition as described in yellow paper
- Instructions are called OPCODES



Solidity

- Most used programming language for Smart Contract Development
- Object-oriented, high-level language for implementing smart contracts
- Solidity is statically typed, supports inheritance, libraries and complex user-defined types.

```
pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {
    uint storedData;

    constructor () {
        storedData = 0;
    }

    function set(uint x) public {
        storedData = x;
    }

    function get() public view returns (uint) {
        return storedData;
    }
}
```

Ether and ERC20 Tokens

- Ether is the native currency of Ethereum Blockchain
- ERC20 is a standard for Fungible Tokens
- ERC20 Tokens are implemented through Smart Contract
- Both Ether and ERC20 represents money in Ethereum blockchain
- Both Ether and ERC20 can be transferred from one account to another
- Developers can implement differently ERC20 functions

```
function name() public view returns (string)
function symbol() public view returns (string)
function decimals() public view returns (uint8)
function totalSupply() public view returns (uint256)
function balanceOf(address _owner) public view returns (uint256 balance)
function transfer(address _to, uint256 _value) public returns (bool success)
function transferFrom(address _from, address _to, uint256 _value) public
  returns (bool success)
function approve(address _spender, uint256 _value) public returns (bool
  success)
function allowance(address _owner, address _spender) public view returns
  (uint256 remaining)
```

Transfer Ether

- Simple transactions form and account to another
- From inside a smart contract:
  - send (2300 gas, returns bool)
  - transfer (2300 gas, throw error)
  - call (forward gas, returns bool)
- Smart contracts must implement receive or fallback

```
pragma solidity 0.8.17;

contract SimpleSender {

  function sendViaSend(address payable bob) public payable {
    bool sent = bob.send(msg.value);
    require(sent, "Failed to send Ether");
  }

  function sendViaTransfer(address payable _bob) public payable {
    _bob.transfer(msg.value);
  }

  function sendViaCall(address payable bob) public payable {
    (bool sent, bytes memory d) = bob.call{value: msg.value}("");
    require(sent, "Failed to send Ether");
  }
}
```

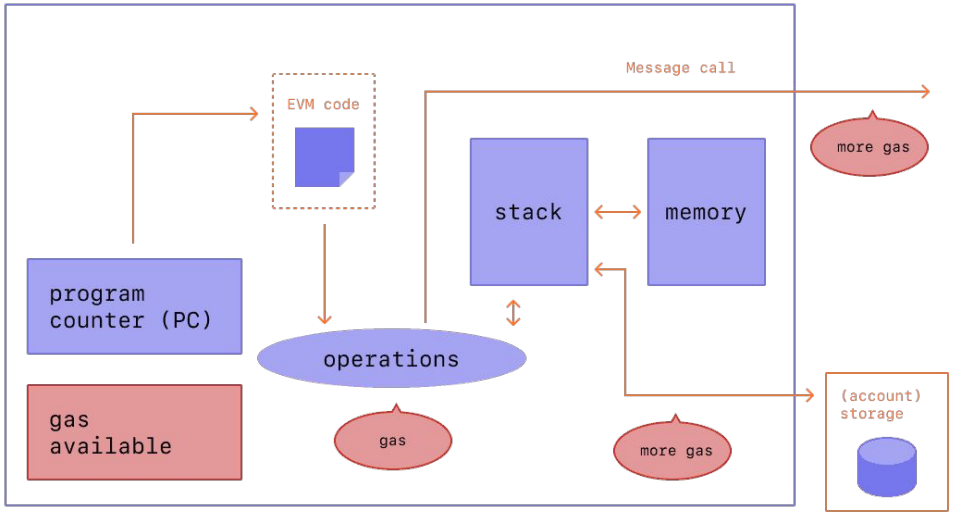
Transfer ERC20

- ERC20 tokens can be transferred only through smart contract interaction
- Transfer vs TransferFrom

```
contract CaFoscariCoin is ERC20 {
...
  mapping(address => uint256) private _balances;
...
  function _transfer(address from,address to,uint256 amt) internal{

    require(from != address(0), "transfer from zero address");
    require(to != address(0), "transfer to zero address");

    uint256 fromBalance = _balances[from];

    require(fromBalance >= amt, "ERC20: amount exceeds balance");
    unchecked {
      _balances[from] = fromBalance - amt;
    }
    _balances[to] += amt;

    emit Transfer(from, to, amt);
  }
...
```

Execution Cost

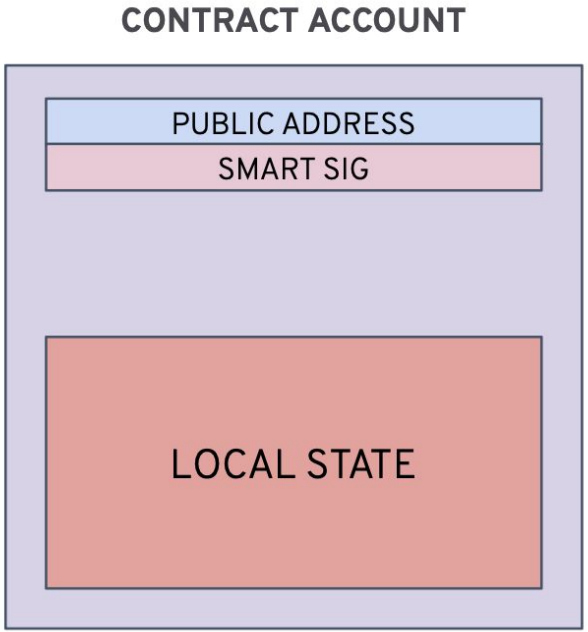
- When EVM executes transactions and smart contracts it consumes gas
- Gas refers to the unit that measures the amount of computational effort required to execute specific operations on the EVM.
- Each OPCODE is assigned a cost.
- Cost is calculated as `units of gas used * (base fee + priority fee)` where `base fee` is a value set by the protocol, `priority fee` is a value set by the user as a tip to the validator



Algorand

Algorand Smart Contracts

- Code executed by the **AVM**
- Written in **Teal/PyTeal** or Reach
- Account type with **AppParams**
- Uniquely identified by **AppId** or **Address**



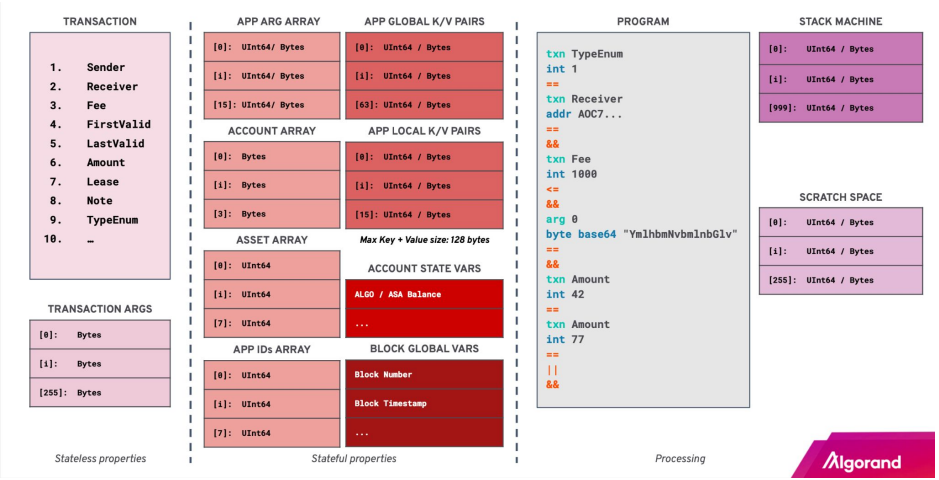
Transactions

In Algorand there are 7 types of transactions:

- Payment
- Key Registration (consensus participation)
- Asset Configuration
- Asset Freeze
- Asset Transfer
- Application Call (both creation and interaction)
- State Proof (consensus tx)

Algorand Virtual Machine

- Contains a stack engine that evaluates smart contracts and smart signatures
- Interprets bytecode created from the compilation of an assembler-like language called TEAL.
- TEAL programs are comprised of a set of operation codes (OPCODES)



Teal/PyTeal

- **TEAL** (Transaction Execution Approval Language) is assembly-like language
- **PyTEAL** is a Python language binding for Algorand Virtual Machine
- Developers can write the contract in Python and then use PyTeal to compile it and produce the TEAL code

```python
from pyteal import *

storedData = Bytes("Stored_data")

handle_creation = Seq(
   App.globalPut(storedData, Int(0)),
   Approve()
)

router = Router(
   "Simple_storage",
   BareCallActions(
       no_op=OnCompleteAction.create_only(handle_creation),
       update_application=OnCompleteAction.always(Reject()),
       delete_application=OnCompleteAction.always(Reject()),
       close_out=OnCompleteAction.never(),
       opt_in=OnCompleteAction.never(),
   ),
)

@router.method
def set(x: abi.Uint64):
   return App.globalPut(storedData, x.get())

@router.method
def get(output: abi.Uint64):
   return output.set(App.globalGet(storedData))

approval_program, clear_program, contract =
router.compile_program(version=6)
```

Algo and ASA

- Algo is the native currency of Algorand blockchain
- Algorand Standard Asset (ASA) is **native (Layer-1)** asset for Fungible (and non-Fungible) tokens
- Both Algo and ASA can be transferred from one account to another
- Both Algo and ASA can represent money in Algorand Blockchain

Transfer ALGO

- Simple payment transaction
- From inside a Smart Contract

```
@router.method
def send_algo(receiver: abi.Account, amount: abi.Uint64):
  payment_cond = And(
    Gtxn[0].type_enum() == TxnType.Payment,
    Gtxn[0].amount() == amount.get(),
    Gtxn[0].receiver() == Global.current_application_address()
  )
  Assert(payment_cond),
  return InnerTxnBuilder.Execute(
      {
        TxnField.fee: Int(0),
        TxnField.type_enum: TxnType.Payment,
        TxnField.receiver: receiver.address(),
        TxnField.amount: amount.get(),
      }
  )
)
```

Transfer ASA

- Simple Asset-transfer transaction
- From inside a Smart Contract
- Remember to opt-in

```
@router.method
def send_asa(
    asa_id: abi.Uint64,
    asset_amount: abi.Uint64,
    asset_sender: abi.Account,
    asset_receiver: abi.Account,
):
 return InnerTxnBuilder.Execute(
    {
      TxnField.fee: Int(0),
      TxnField.type_enum: TxnType.AssetTransfer,
      TxnField.xfer_asset: asa_id,
      TxnField.asset_amount: asset_amount,
      TxnField.asset_sender: asset_sender,
      TxnField.asset_receiver: asset_receiver,
    }
 )
```

Execution Cost

- Fees for transactions on Algorand are set as a function of network congestion and based on the size in bytes of the transaction. Every transaction must at least cover the minimum fee (1000μA or 0.001A).
- fee = max(current_fee_per_byte*len(txn_in_bytes), min_fee)

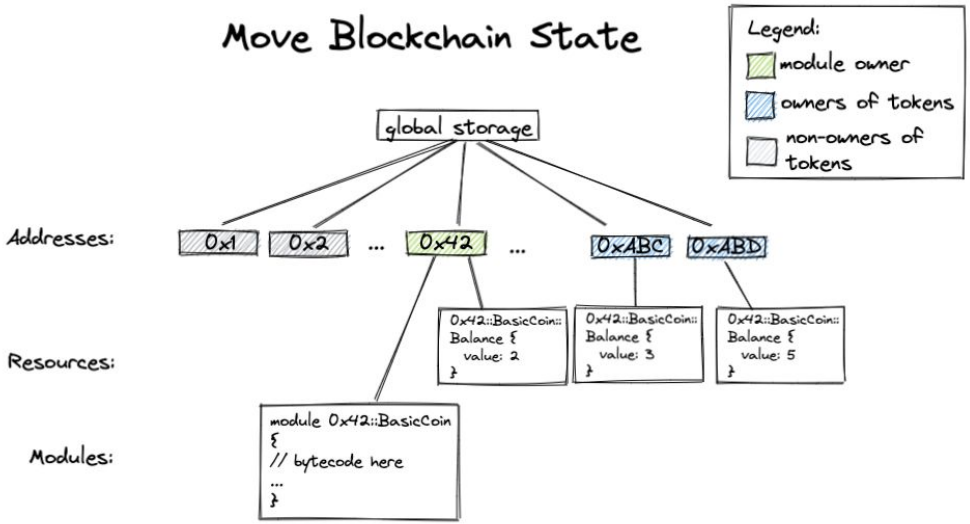
Other works

- Smart signatures
- Smart ASA

Aptos

Aptos Smart Contracts

- Code executed by the MoveVM
- Written in Move Language
- Stored as a Module inside an Account
- Uniquely identified by Account address + Module name



Transactions

Aptos transactions contain information such as the sender's account address, authentication from the sender, the desired operation to be performed on the Aptos blockchain, and the amount of gas the sender is willing to pay to execute the transaction.

- Entry point
- Script

- `Signature`: a digital signature to verify the transaction.
- `Sender address`: The sender's account address.
- `Sender public key`: The public authentication key..
- `Program`: which comprises:
  - A Move module and function name or a move bytecode transaction script.
  - An optional list of inputs to the script.
  - An optional list of Move bytecode modules to publish.
- `Gas price` (in specified gas units): This is the amount the sender is willing to pay per unit of gas to execute the transaction.
- `Maximum gas amount`: The maximum gas amount is the maximum gas units the transaction is allowed to consume.
- `Sequence number`: an unsigned integer.
- Expiration time: A timestamp after which the transaction ceases to be valid (i.e., expires).

Move Language

- Language for secure, sandboxed, and formally verified programming
- Originally developed for Libra Blockchain, later renamed as Diem Blockchain (Facebook)
- Takes its cue from Rust by using resource types with **move** semantics

```
module deploy_address::set_val {

  use std::signer;

  struct Test has key {
    test_val: u64
  }

  public entry fun set_val(acc: &signer, n: u64)acquires Test {
    let account_addr = signer::address_of(acc);

    if (!exists<Test>(account_addr)) {
      move_to(acc, Test { test_val: n });
    }

    else {
      let test = borrow_global_mut<Test>(account_addr);
      test.test_val = n;
    }
  }
}
```

Move Resources

- Used for defining assets inside the blockchain, such as Coins or Tokens
- Stored within individual accounts
- Defined as Structs with Key ability
- The Move type system provides special safety guarantees for resources.
- Move resources can never be duplicated, reused, or discarded.
- A resource type can only be created or destroyed by the module that defines the type.
- These guarantees are enforced statically by the Move virtual machine via bytecode verification. The Move virtual machine will refuse to run code that has not passed through the bytecode verifier

```
module M {
    struct T has key, store {
      field: u8
    }
}
```

Future works

Thank you for your attention

References

- https://ethereum.org/en/developers/docs/evm/
- https://ethereum.github.io/yellowpaper/paper.pdf
- https://docs.soliditylang.org/en/latest/

- https://www.sciencedirect.com/science/article/pii/S030439751930091X
- https://developer.algorand.org/docs/
- https://github.com/algorand/go-algorand/blob/master/data/basics/userBalance.go

- https://developers.diem.com/papers/diem-move-a-language-with-programmable-resources/2019-06-18.pdf
- https://arxiv.org/abs/2004.05106
- https://move-language.github.io/move/introduction.html
- https://move-book.com/index.html
- https://aptos.dev/