Rafael Fernández Ortiz

Assignments 2

Exercise 1

Given the following contract:

```
pragma solidity ^0.4.0;

contract exercise1 {
    function fn(uint n) external pure returns (uint) {
        if (n < 10) { return n*n ; }
        else { return n/2 ; }
    }
}
```

Answer the following questions:

1. Generate the assembly code of the contract and save it on a text file.
2. Determine the constructor code and the runtime code (specify line numbers in the text file).
3. Determine the blocks that compose the runtime code of the program (specify line numbers and begin-end instructions).

1.1 Generate the assembly code of the contract and save it on a text file

The assembly code of the contract is allocated in `opcode-exercise-2-1.txt`

1.2 Determine the constructor code and the runtime code and 1.3 Determine the blocks that compose the runtime code of the program

The constructor code starts in line 1 (beginning with `PUSH1 0x80`, `PUSH1 0x40` and `MSTORE` opcode) and ending in line 21 (with `STOP` opcode). After that, runtime code starts in line 22 (beginning with `PUSH1 0x80`, `PUSH1 0x40` and `MSTORE` opcode after `STOP` opcode (line 21)) and ending in line 145 with `STOP` opcode.

Exercise 2

Given the following contract:

```
pragma solidity ^0.4.0;

contract exercise2 {
    uint[] arr = new uint[](5);
    function powers() external {
        for (uint i = 0; i < arr.length; i++)
            { arr[i] = i**i; }
    }
}
```

Answer the following questions:

1. Consider the **runtime code only**
2. Draw **CFG** of the runtime code (label nodes with tags)

3. Locate the instructions for accessing storage. Identify the purpose of each storage access instruction (`SLOAD, SSTORE`). Add this information to the nodes of the CFG

4. (Optional) Could we change the Solidity code to reduce the number of accesses to storage? How?

2.1 Consider the runtime code only and 2.2 draw CFG of the runtime code (label nodes with tags)

The assembly code of the contract is allocated in opcode-exercise-2-2.txt

```
{0}: [PUSH1 0x80,PUSH1 0x40,MSTORE,PUSH1 0x4,CALLDATASIZE,LT,PUSH1 0x3F,JUMPI]
{1}: [PUSH1 0x0,CALLDATALOAD,PUSH29 0
    x100000000000000000000000000000000000000000000000000000000,SWAP1,DIV,PUSH4 0
    xFFFFFFFF,AND,DUP1,PUSH4 0x72A20C78,EQ,PUSH1 0x44,JUMPI]
{2}: [JUMPDEST,PUSH1 0x0,DUP1,REVERT]
{3}: [JUMPDEST,CALLVALUE,DUP1,ISZERO,PUSH1 0x4F,JUMPI]
{4}: [PUSH1 0x0,DUP1,REVERT]
{5}: [JUMPDEST,POP,PUSH1 0x56,PUSH1 0x58,JUMP]
{6}: [JUMPDEST,STOP]
{7}: [JUMPDEST,PUSH1 0x0,DUP1,SWAP1,POP,JUMPDEST,PUSH1 0x0,DUP1,SLOAD,SWAP1,POP,DUP2,
    LT,ISZERO,PUSH1 0x95,JUMPI]
{8}: [DUP1,DUP2,EXP,PUSH1 0x0,DUP3,DUP2,SLOAD,DUP2,LT,ISZERO,ISZERO,PUSH1 0x7B,JUMPI]
{9}: [INVALID]
{10}: [JUMPDEST,SWAP1,PUSH1 0x0,MSTORE,PUSH1 0x20,PUSH1 0x0,KECCAK256,ADD,DUP2,SWAP1,
    SSTORE,POP,DUP1,DUP1,PUSH1 0x1,ADD,SWAP2,POP,POP,PUSH1 0x5E,JUMP]
{11}: [JUMPDEST,POP,JUMP]
{12}: [STOP]
{13}: [LOG1,PUSH6 0x627A7A723058,KECCAK256 0xad,XOR,EXP 0xf5,PUSH14 0
    xB73AA6BA3E26CA16BBDC0070FBCA,MLOAD 0xcb,PUSH31 0
    xE1F0C63871E4002900000000000000000000000000000000000000000000000000]
```

```
0 --> [1,2]
1 --> [2,3]
2 --> END
3 --> [4,5]
4 --> END
5 --> [7]
6 --> END
7 --> [8,10]
8 --> [9,10]
9 --> END
10 --> [11]
11 --> SYMBOLIC (not resolved)
12 --> END
13 --> END
```

2.3 Locate the instructions for accessing storage. Identify the purpose of each storage access instruction (`SLOAD, SSTORE`). Add this information to the nodes of the CFG.

- `SLOAD` opcode in exercise2 is allocated in lines: 49 and 63. Its purpose is load the array and the value that the array points to at the index i.

- `SSTORE` opcode in exercise2 is allocated in line: 81. Its purpose is to store the new value calculated into the array and to point to that value at index i.

2.4 (Optional) Could we change the Solidity code to reduce the number of accesses to storage? How?

If store the length of array `arr` in a global variable, It will be more efficient, because It doesn't have to consult its length.

Exercise 3

Given the following code:

```
pragma solidity ^0.4.0;

contract exercise3 {
    uint [] arr = new uint[](5) ;
    function p1() external view returns (uint) {
        uint sumEven = 0;
        for (uint i = 0; i < arr. length ; i +=2) {
            sumEven += arr[i];
        }
        return sumEven;
    }
    function p2() external view { uint [] memory local = arr; }
    function p3() external view { uint [] storage local = arr; }
}
```

- Load into `Gastap` the following contract and analyze it.
- Explain the results obtained for these functions.

The `Gastap` outputs for the three functions are:

```
function p1()
Memory UB for p1():  15
Opcodes UB for p1():  1092+2559*nat(arr/2+1/2)
```

The memory cost is constant because it just store only one variable. However, the instruction is executed as many times as the length of the array divided by 2 (because it increment i variable by 2 on every iteration).

```
function p2()
Memory UB for p2():  3*max([nat(arr)+4+2])+pow(max([nat(arr)+4+2]),2)/512
Opcodes UB for p2():  2784+850*nat(arr-1/32)
```

???

```
function p3()
Memory UB for p3():  9
Opcodes UB for p3():  137
```

In this case, the function create a reference to `arr`, therefore the cost is constant for memory and for opcodes.

References

[1] Repo. Github - https://github.com/rafafrdz/ethereum-control-flow-graph-bytecode

[2] Contro, F., Crosara, M., Ceccato, M. y Dalla Preda, M., *EtherSolve: Computing an Accurate Control-Flow Graph from Ethereum Bytecode* - https://arxiv.org/pdf/2103.09113.pdf