Introducción a la tecnología blockchain y smart contracts

Práctica 6

Javier Mulero Martín y Jorge del Valle Vázquez

19 de mayo de 2022

En esta práctica vamos a aprender a interpretar el Bytecode EVM, cómo controlar el consumo de gas y aprender el uso de la programación con Yul en los contratos.

1. Bytecode EVM, consumo de gas

1. Genera el código ensamblador del contrato[1] y guárdalo en un fichero de texto:
El código ensamblador esta adjunto en el fichero *bytecode.txt*.

2. Dibuja el grado CFG de `computeSum()`:
El grafo se muestra en la figura 1. Está construido desde el `tag 0`, aunque realmente el comienzo de la función `computeSum()` está en el `tag 3`.

3. Localiza las instrucciones que acceden a *storage*:

Instrucción	Nodo	Expresión	Objetivo
SSTORE	9	storage[1] = 0	La variable sum (en el slot 1 del storage) se inicia a 0
SLOAD	16	value = storage[0]	Pone en la cima de la pila (value) la longitud de arr (en slot 0 de storage)
SLOAD	16b	value = storage[0]	Pone en la cima la longitud de arr, para comprobar que no se sale de rango
SLOAD	19	value = arr[i]	Pone en la cima arr[i]
SLOAD	19	value = sum	Pone en la cima sum
SSTORE	19	storage[1] = sum + arr[i]	Guarda en sum la suma sum + arr[i]

4. Modifica el códifo de `computeSum()` con código Solidity para dar una versión mejorada con un consumo mínimo de gas:
La mejora, mostrada en la figura 2, se basa en 3 ideas:

- La primera, utilizamos una variable local `aux` para acumular la suma, en lugar de estar acumulando directamente en `sum` que se encuentra en *storage* y cada vuelta del bucle hacíamos un SLOAD y un SSTORE. En este caso solo haremos un SSTORE al final del bucle para almacenar la suma.

[1] El contrato se encuentra en el Anexo A

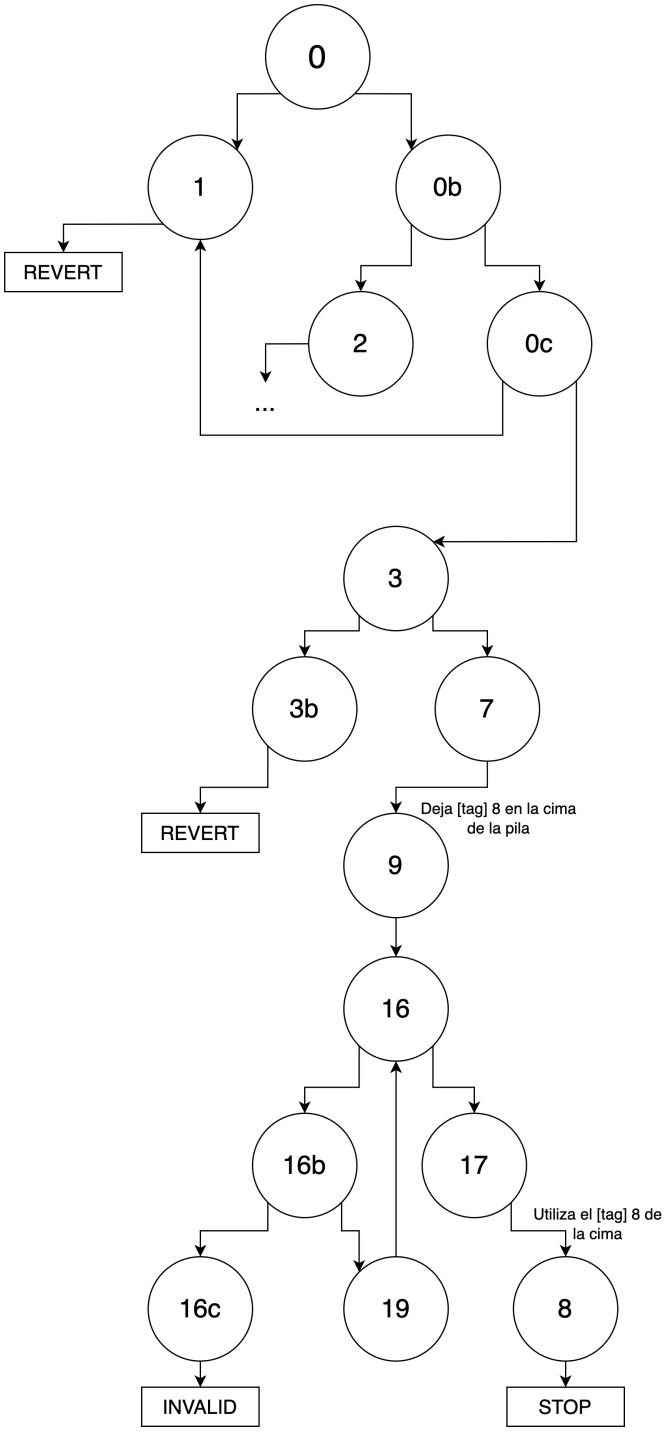

Figura 1: Grafo CFG

```
1     function computeSum() external {
2             uint[] memory c = arr;
3             uint aux = 0;
4             uint l = arr.length;
5             for (uint i = 0; i < l; i++) {
6                 aux = aux + c[i];
7             }
8             sum = aux;
9         }
```

Figura 2: Mejora de la función `computeSum()` con código Solidity

- La segunda, utilizamos otra variable local para almacenar la longitud de `arr`, ya que previamente en cada vuelta del bucle accedíamos a *storage* (slot 1) donde se almacena la longitud del array. De esta manera, ahora solo accedemos una vez antes de iniciar el bucle.
- La tercera, hacemos una copia en *memory* del array, de modo que en cada vuelta del bucle ya no accedemos a *storage*, sino a *memory*, ya que el compilador de solidity realiza la copia en *memory* de una manera más eficiente que el acceso que hacíamos anteriormente.

De esta forma, el consumo de gas queda de la siguiente forma:

- **Con mejora**: 274322
- **Original**: 312481

2. Programación con Yul y el *assembly block*

5. Programa una función `maxMinMemory(uint[] memory arr)` que calcule la distancia entre el máximo y el mínimo del array utilizando exclusivamente código Yul:

El código se muestra en la figura 3. En *memory*, el array apunta en primera posición a su longitud, y a partir de ahí, se encuentran los elementos (cada uno es una palabra), que los recorremos hasta llegar al último del array.

Para probarlo, introducimos un array aleatorio, como por ejemplo $[40, 32, 1, 76, 23, 87, 100, 3]$, y vemos que el resultado que muestra es 99.

6. Realiza lo mismo que en el apartado anterior para un array en *storage*, utilizando exclusivamente código Yul:

El código para este apartado se muestra en la figura 4. La idea es que el `slot` del array es el 0, donde se encuentra la longitud del array. Despúes guardamos número de `slot`(0) en la posición reservada `0x0` de *memory* para luego, usando la función **keccak256** (de `0x0` a `0x20`, *hash* del slot 1 que indica el comienzo de los datos), realizar el *hash* para acceder a la primera posición del array. Después cargamos el primer valor para el máximo y el

```
1
2     contract lab6ex5 {
3         function maxMinMemory(uint[] memory arr) public pure returns (uint
              maxmin) {
4             assembly {
5                 function fmaxmin (slot) -> maxVal, minVal{
6                     let len := mload(slot)
7                     let data := add(slot, 0x20)
8                     maxVal := mload(data)
9                     minVal := maxVal
10                    let i := 1
11                    for {} lt(i,len) {i:= add(i,1)}
12                    {
13                        let elem := mload(add(data,mul(i,0x20)))
14                        if gt(elem,maxVal) { maxVal := elem }
15                        if lt(elem,minVal) { minVal := elem }
16                    }
17                }
18                let resultmax,resultmin := fmaxmin(arr)
19                maxmin:=sub(resultmax,resultmin)
20            }
21        }
22    }
```

Figura 3: `fmaxmin` en código Yul para array en *memory*

mínimo, y recorremos el resto (de `slot` en `slot`) haciendo las comprobaciones necesarias para encontrar el mínimo y máximo del array.

```
contract lab6ex6 {
    uint[] public arr;
    function generate(uint n) external {
        // Populates the array with some weird small numbers.
        bytes32 b = keccak256("seed");
        for (uint i = 0; i < n; i++) {
            uint8 number = uint8(b[i % 32]);
            arr.push(number);
        }
    }
    function maxMinStorage() public view returns (uint maxmin){
        assembly {
            function fmaxmin (slot) -> maxVal, minVal{
                let len := sload(slot)
                mstore(0x0,slot)
                let data := keccak256(0x0, 0x20)
                maxVal := sload(data)
                minVal := maxVal
                let i := 1
                for {} lt(i,len) {i:= add(i,1)}
                {
                    let elem := sload(add(data,i))
                    if gt(elem,maxVal) { maxVal := elem }
                    if lt(elem,minVal) { minVal := elem }
                }
            }
            let resultmax,resultmin := fmaxmin(arr.slot)
            maxmin:=sub(resultmax,resultmin)
        }
    }
}
```

Figura 4: `fmaxmin` en código Yul para array en *storage*

7. Crea otro contrato de `maxMinStorage()` en Solidity y compara el coste de ejecución respecto de la versión en Yul. Analiza los resultados obtenidos.

El código para este apartado se muestra en la figura 5. Podemos observar dos versiones distintas.

En la primera, la causa principal de la reducción del consumo de gas se debe a la disminución de accesos a *storage*, principalmente los accesos a la longitud del array que suceden en cada iteración para la comprobación de terminación del bucle, que en la implementación de código Yul se hace solo una vez (`let len := sload (slot)`).

En cambio, en la segunda, hacemos una optimización para ahorrarnos estos accesos a *storage*, y vemos cómo aún así, la versión con código Yul tiene un menor coste de ejecución. Esto nos indica que al utilizar código Yul, al ser en más bajo nivel, controlamos de forma más directa los accesos a *storage*, mientras que en código Solidity hace más llamadas para controlar, por ejemplo, que el índice de acceso al array no se salga, y consume de esta forma más gas.

De esta forma, el consumo de gas queda de la siguiente forma:

- **Yul con array en storage**: 245026
- **Original**: 314794
- **Original "optimizado"**: 276002

```
1   // Original (no optimizado)
2   function maxMinStorage() public view returns (uint maxmin){
3       uint maxVal=arr[0];
4       uint minVal=arr[0];
5       for (uint i = 1; i < arr.length; i++) {
6           if(maxVal<arr[i]){
7               maxVal= arr[i];
8           }
9           if(minVal>arr[i]){
10              minVal= arr[i];
11          }
12      }
13      maxmin=maxVal-minVal;
14  }
15  // Original optimizado
16  function maxMinStorage() public view returns (uint maxmin){
17      uint maxVal=arr[0];
18      uint minVal=maxVal;
19      uint l = arr.length;
20      for (uint i = 1; i < l; i++) {
21          uint elem = arr[i];
22          if(maxVal<elem){
23              maxVal= elem;
24          }
25          if(minVal>elem){
26              minVal= elem;
27          }
28      }
29      maxmin=maxVal-minVal;
30  }
```

Figura 5: `fmaxmin` en código Solidity para array en *storage*

Anexos

A. Contratos

```
1  // SPDX-License-Identifier: GPL-3.0
2
3  pragma solidity ^0.8.0;
4
5  contract lab6ex1 {
6      uint[] arr;
7      uint sum;
8      function generate(uint n) external {
9          for (uint i = 0; i < n; i++) {
10             arr.push(i*i);
11         }
12     }
13     function computeSum() external {
14         sum = 0;
15         for (uint i = 0; i < arr.length; i++) {
16             sum = sum + arr[i];
17         }
18     }
19 }
20
21 contract lab6ex4 {
22     uint[] arr;
23     uint sum;
24     function generate(uint n) external {
25         for (uint i = 0; i < n; i++) {
26             arr.push(i*i);
27         }
28     }
29     function computeSum() external {
30         uint [] memory c = arr ;
31         uint aux = 0;
32         uint l = arr.length ;
33         for ( uint i = 0; i < l ; i ++) {
34             aux = aux + c [ i ];
35         }
36         sum = aux ;
37     }
38 }
39
40 contract lab6ex5 {
41     function maxMinMemory(uint[] memory arr) public pure returns (uint
       maxmin) {
42         assembly {
43             function fmaxmin (slot) -> maxVal, minVal{
44                 let len := mload(slot)
```

```
                let data := add(slot, 0x20)
                maxVal := mload(data)
                minVal := maxVal
                let i := 1
                for {} lt(i,len) {i:= add(i,1)}
                {
                    let elem := mload(add(data,mul(i,0x20)))
                    if gt(elem,maxVal) { maxVal := elem }
                    if lt(elem,minVal) { minVal := elem }
                }
            }
            let resultmax,resultmin := fmaxmin(arr)
            maxmin:=sub(resultmax,resultmin)
        }
    }
}
contract lab6ex6 {
    uint[] public arr;
    function generate(uint n) external {
        // Populates the array with some weird small numbers.
        bytes32 b = keccak256("seed");
        for (uint i = 0; i < n; i++) {
            uint8 number = uint8(b[i % 32]);
            arr.push(number);
        }
    }
    function maxMinStorage() public view returns (uint maxmin){
        assembly {
            function fmaxmin (slot) -> maxVal, minVal{
                let len := sload(slot)
                mstore(0x0,slot)
                let data:=keccak256(0x0, 0x20)
                maxVal := sload(data)
                minVal := maxVal
                let i := 1
                for {} lt(i,len) {i:= add(i,1)}
                {
                    let elem := sload(add(data,i))
                    if gt(elem,maxVal) { maxVal := elem }
                    if lt(elem,minVal) { minVal := elem }
                }
            }
            let resultmax,resultmin := fmaxmin(arr.slot)
            maxmin:=sub(resultmax,resultmin)
        }
    }
}
contract lab6ex7 {
    uint[] public arr;
    function generate(uint n) external {
        // Populates the array with some weird small numbers.
```

```
96          bytes32 b = keccak256("seed");
97          for (uint i = 0; i < n; i++) {
98              uint8 number = uint8(b[i % 32]);
99              arr.push(number);
100         }
101     }
102     function maxMinStorage() public view returns (uint maxmin){
103         uint maxVal=arr[0];
104         uint minVal=maxVal;
105         for (uint i = 1; i < arr.length; i++) {
106             if(maxVal<arr[i]){
107                 maxVal= arr[i];
108             }
109             if(minVal>arr[i]){
110                 minVal= arr[i];
111             }
112         }
113         maxmin=maxVal-minVal;
114     }
115 }
116 
117 contract lab6ex7_2 {
118     uint[] public arr;
119     function generate(uint n) external {
120         // Populates the array with some weird small numbers.
121         bytes32 b = keccak256("seed");
122         for (uint i = 0; i < n; i++) {
123             uint8 number = uint8(b[i % 32]);
124             arr.push(number);
125         }
126     }
127 
128     function maxMinStorage2() public view returns (uint maxmin){
129         uint maxVal=arr[0];
130         uint minVal=maxVal;
131         uint l = arr.length;
132         for (uint i = 1; i < l; i++) {
133             uint elem = arr[i];
134             if(maxVal<elem){
135                 maxVal= elem;
136             }
137             if(minVal>elem){
138                 minVal= elem;
139             }
140         }
141         maxmin=maxVal-minVal;
142     }
143 
144 }
```