Less Gas, More Fun: Optimising Smart Contracts through Yul

Christian Reitwiessner

devcon4 - Prague - 2018-10-31

Christian Reitwiessner
Ethereum Foundation
@ethchris github.com/chriseth chris@ethereum.org

https://chriseth.github.io/notes/talks/solidity_optimizer_devcon4/

Definition: Optimizer

Software that transforms a program into a program that

Definition: Optimizer

Software that transforms a program into a program that

1. requires fewer resources (or at least not more)

Definition: Optimizer

Software that transforms a program into a program that

1. requires fewer resources (or at least not more)
2. is semantically equivalent.

Definition: Optimizer

Software that transforms a program into a program that

1. requires fewer resources (or at least not more)
2. is semantically equivalent.

Example:

```
contract C {
    function f(uint x) public pure returns (uint) {
        return x**2;
    }
}
```

Definition: Optimizer

Software that transforms a program into a program that

1. requires fewer resources (or at least not more)
2. is semantically equivalent.

Example:

```
contract C {
    function f(uint x) public pure returns (uint) {
        return x**2;
    }
}
```

```
contract C {
    function f(uint x) public pure returns (uint) {
        return x*x; // cheaper in EVM
    }
}
```

Problem 1

Which resources to optimize?

Tradeoff: code size (deploy costs) <-> runtime

Different tradeoff for popular functions or inside loops.

Problem 2

Even with clearly defined metric, perfect optimizer is theoretically impossible.

Reason: Semantic equivalence

Problem 2

Even with clearly defined metric, perfect optimizer is theoretically impossible.

Reason: Semantic equivalence

"Proof:" Halting problem: There is no program that decides whether a given program halts (reverts) on all inputs.

Problem 2

Even with clearly defined metric, perfect optimizer is theoretically impossible.

Reason: Semantic equivalence

"Proof:" Halting problem: There is no program that decides whether a given program halts (reverts) on all inputs.

If perfect optimizer exists, it can be used to create such a program:

Problem 2

Even with clearly defined metric, perfect optimizer is theoretically impossible.

Reason: Semantic equivalence

"Proof:" Halting problem: There is no program that decides whether a given program halts (reverts) on all inputs.

If perfect optimizer exists, it can be used to create such a program:

A program reverts on all inputs if and only if the optimizer's output for it is (similar to) "contract C {}".

Why an Optimizer?

- Cheaper Smart Contracts

Why an Optimizer?

- Cheaper Smart Contracts
- More modular and understandable code

Why an Optimizer?

- Cheaper Smart Contracts
- More modular and understandable code

```
contract Voting {
    // ...
    function weightOf(address _voter) internal view returns (uint) {
        return _voter == m_owner ? 10 : 1;
    }
    function vote(Outcome memory _o) public {
        require(!hasVoted(msg.sender));
        m_votes[_o] = weightOf(msg.sender);
    }
}
```

Outline of Rest of Talk

- Description of current Solidity optimizer
- Yul-based optimizer, current status and future plans

Current Solidity Optimizer

Everything based on "sugared" EVM opcode stream:

- peephole optimizer
- deduplicator
- dead code removal
- constant optimizer
- common subexpression eliminator (much more than that)

Solidity's CSE Optimizer

Solidity's CSE Optimizer

- chop code into blocks

Solidity's CSE Optimizer

- chop code into blocks
- feed to component opcode-by-opcode

Solidity's CSE Optimizer

- chop code into blocks
- feed to component opcode-by-opcode
- component builds symbolic expression trees

Solidity's CSE Optimizer

- chop code into blocks
- feed to component opcode-by-opcode
- component builds symbolic expression trees
- expression trees are simplified according to rules like `7 + (x + 2) -> x + 9` and `(x == x) -> 1`.

Solidity's CSE Optimizer

- chop code into blocks
- feed to component opcode-by-opcode
- component builds symbolic expression trees
- expression trees are simplified according to rules like
  `7 + (x + 2) -> x + 9` and
  `(x == x) -> 1`.
- changes to memory and storage are recorded (abstractly)

Solidity's CSE Optimizer

```
push 0          // -> $1: "0"
calldataload    // -> $2: calldataload($1)
sload           // -> $3: sload($2)
push 1          // -> $4: "1"
sstore          // storage[1] = storage[calldata[0]]

push 0          // $1 again!
calldataload    // calldataload($1) -> $2 again!
sload           // $3 again! (important caveat!)
push 1          // $4 again!
sload           // no storage changes, so still
                // storage[calldata[0]], i.e. $3
eq              // $3 = $3? true!
```

Solidity's CSE Optimizer

Solidity's CSE Optimizer

- code-re-generation unit re-builds code from bottom to top, hopefully more efficient

Solidity's CSE Optimizer

- code-re-generation unit re-builds code from bottom to top, hopefully more efficient
- main benefits: eliminate redundant stores, do not re-compute same expressions, remove dead code

Solidity's CSE Optimizer

- code-re-generation unit re-builds code from bottom to top, hopefully more efficient
- main benefits: eliminate redundant stores, do not re-compute same expressions, remove dead code
- main drawbacks: very opaque, only very local, no notion of functions (no inlining) or loops

Yul

New intermediate language for Solidity. Structured language, can compile to EVM and wasm. Extremely simple syntax. See talk at devcon3.

```
{
    function owner() -> o {
        o := sload(0)
    }
    function transfer(recipient, amount) -> s {
        s := call(0, recipient, amount, 0, 0, 0, 0)
    }
    // withdraw
    if eq(caller(), owner()) {
        switch transfer(caller(), balance())
        case 0 { revert(0, 0) }
        case 1 { stop() }
    }
}
```

Yul-based Optimizer

- Sequence of small and local modifications
- Readable text export and re-import possible at every step
- No gathering of data in opaque data structures
- Keep structure (functions, loop, no "goto"), direct translation to wasm.

Yul-based Optimizer

Tricky part:

When to call which component?

This will only impact efficiency! As long as each step is correct in isolation, whole sequence will be correct.

```
function array_sum(x) -> sum {
    let length := mload(x)
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        sum := add(sum, array_load(x, i))
    }
}
function array_load(x, i) -> v {
    let len := mload(x)
    if iszero(lt(i, len)) { revert() }
    let data := add(x, 0x20)
    v := mload(add(data, mul(i, 0x20)))
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        let _1 := array_load(x, i)
        sum := add(sum, _1)
    }
}
function array_load(x, i) -> v {
    let len := mload(x)
    if iszero(lt(i, len)) { revert() }
    let data := add(x, 0x20)
    v := mload(add(data, mul(i, 0x20)))
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        let v
        {
            let len := mload(x)
            if iszero(lt(i, len)) { revert() }
            let data := add(x, 0x20)
            v := mload(add(data, mul(i, 0x20)))
        }
        let _1 := v
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {


            let len := mload(x)
            if iszero(lt(i, len)) { revert() }
            let data := add(x, 0x20)
            let _1 := mload(add(data, mul(i, 0x20)))


        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        let len := mload(x)
        if iszero(lt(i, len)) { revert() }
        let data := add(x, 0x20)
        let _1 := mload(add(data, mul(i, 0x20)))
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)
    let len := mload(x)
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {

        if iszero(lt(i, len)) { revert() }
        let _1 := mload(add(data, mul(i, 0x20)))
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)

    for { let i := 0 } lt(i, length) { i := add(i, 1) } {

        if iszero(lt(i, length)) { revert() }
        let _1 := mload(add(data, mul(i, 0x20)))
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)

    for { let i := 0 } lt(i, length) { i := add(i, 1) } {

        if iszero(1) { revert() }
        let _1 := mload(add(data, mul(i, 0x20)))
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)

    for { let i := 0 } lt(i, length) { i := add(i, 1) } {

        if 0 { revert() }
        let _1 := mload(add(data, mul(i, 0x20)))
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)

    for { let i := 0 } lt(i, length) { i := add(i, 1) } {


        let _1 := mload(add(data, mul(i, 0x20)))
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)

    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        let _2 := mul(i, 0x20)
        let _3 := add(data, _2)
        let _1 := mload(_3)
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)
    let _2 := -0x20
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        _2 := add(_2, 0x20)
        let _3 := add(data, _2)
        let _1 := mload(_3)
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)
    let _2 := add(-0x20, data)
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        _2 := add(_2, 0x20)

        let _1 := mload(_2)
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)
    let data := add(x, 0x20)
    let _2 := x
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        _2 := add(_2, 0x20)

        let _1 := mload(_2)
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)

    let _2 := x
    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        _2 := add(_2, 0x20)

        let _1 := mload(_2)
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)


    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        x := add(x, 0x20)

        let _1 := mload(x)
        sum := add(sum, _1)
    }
}
```

```
function array_sum(x) -> sum {
    let length := mload(x)


    for { let i := 0 } lt(i, length) { i := add(i, 1) } {
        x := add(x, 0x20)


        sum := add(sum, mload(x))
    }
}
```

Memory Optimizations

Memory in EVM is short-lived, so no memory management.

Optimizer can avoid allocation of temporary memory.

Introduce "memory objects" as first-class citizens into Yul and track their lifetime.

Small temporary memory objects can even be allocated in scratch space.

Summary

New optimizer will be safer, more transparent and more powerful.

Main challenge: Find good heuristics. Code size very important, but intermediate code can be large.