SMT-based Verification of Solidity Smart Contracts

Leonardo Alt, Christian Reitwiessner

ISoLA - Limassol, Cyprus - 2018-11-05

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

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

Existing Formal Verification Projects

Existing Formal Verification Projects

- EVM Formal Semantics: Eth-Isabelle, K-EVM, Ethereum-Lem, …

Existing Formal Verification Projects

- EVM Formal Semantics: Eth-Isabelle, K-EVM, Ethereum-Lem, …
- EVM Bytecode Symbolic Execution: Oyente, Mythril, Mayan, Securify, …

Existing Formal Verification Projects

- EVM Formal Semantics: Eth-Isabelle, K-EVM, Ethereum-Lem, …
- EVM Bytecode Symbolic Execution: Oyente, Mythril, Mayan, Securify, …
- Translation of Solidity to Verifiable Languages: Why3, F*, ZEUS, K-Solidity

Existing Formal Verification Projects

- EVM Formal Semantics: Eth-Isabelle, K-EVM, Ethereum-Lem, …
- EVM Bytecode Symbolic Execution: Oyente, Mythril, Mayan, Securify, …
- Translation of Solidity to Verifiable Languages: Why3, F*, ZEUS, K-Solidity
- Our Approach: SMT-based Bounded Model Checker

Goals

- automatic verification as part of the compiler stack
- minimal effort by the programmer
- no verification conditions
- no additional tools to install

Goals

- automatic verification as part of the compiler stack
- minimal effort by the programmer
- no verification conditions
- no additional tools to install
- automatic counterexamples

Goals

- automatic verification as part of the compiler stack
- minimal effort by the programmer
- no verification conditions
- no additional tools to install
- automatic counterexamples
- practicality over completeness

Goals

- automatic verification as part of the compiler stack
- minimal effort by the programmer
- no verification conditions
- no additional tools to install
- automatic counterexamples
- practicality over completeness

first and automatic helper, more thorough and sophisticated analysis based on EVM bytecode

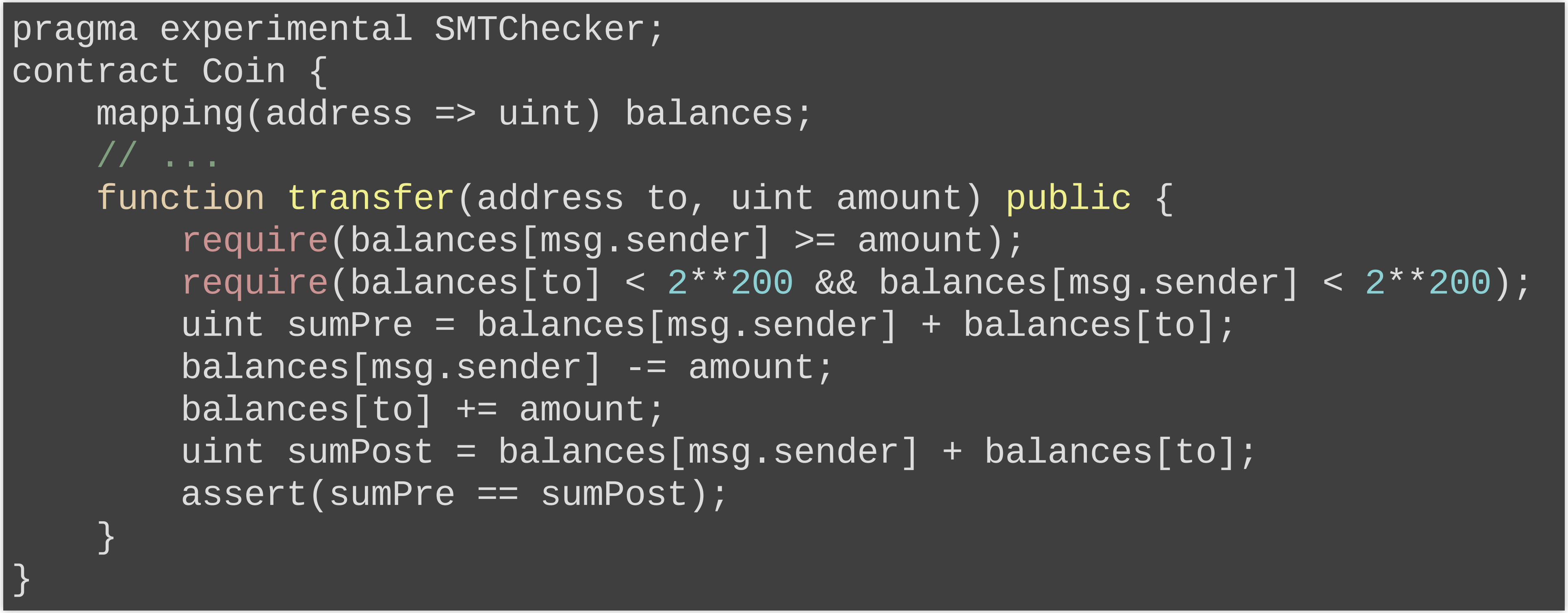
Verification Targets

- arithmetic overflow / underflow
- division by zero
- trivial conditions / unreachable code
- assertions

```
pragma experimental SMTChecker;
contract Coin {
    mapping(address => uint) balances;
    // ...
    function transfer(address to, uint amount) public {


        // Error: Underflow for balances[msg.sender] = 0 and amount = 1
        balances[msg.sender] -= amount;
        balances[to] += amount;


    }
}
```

```
pragma experimental SMTChecker;
contract Coin {
    mapping(address => uint) balances;
    // ...
    function transfer(address to, uint amount) public {
        require(balances[msg.sender] >= amount);


        balances[msg.sender] -= amount;
        balances[to] += amount;
        // Error: overflow for balances[to] = 2**256-1 and amount = 1

    }
}
```

```
pragma experimental SMTChecker;
contract Coin {
    mapping(address => uint) balances;
    // ...
    function transfer(address to, uint amount) public {
        require(balances[msg.sender] >= amount);
        require(balances[to] < 2**200 && balances[msg.sender] < 2**200);

        balances[msg.sender] -= amount;
        balances[to] += amount;


    }
}
```



```
pragma experimental SMTChecker;
contract Coin {
    mapping(address => uint) balances;
    // ...
    function transfer(address to, uint amount) public {
        require(balances[msg.sender] >= amount);
        require(balances[to] < 2**200 && balances[msg.sender] < 2**200);
        uint sumPre = balances[msg.sender] + balances[to];
        balances[msg.sender] -= amount;
        balances[to] += amount;
        uint sumPost = balances[msg.sender] + balances[to];
        assert(sumPre == sumPost);
    }
}
```

How is it done?

How is it done?

- traverse AST in execution order

How is it done?

- traverse AST in execution order
- introduce variable for each expression and new assignment

How is it done?

- traverse AST in execution order
- introduce variable for each expression and new assignment
- collect constraints

How is it done?

- traverse AST in execution order
- introduce variable for each expression and new assignment
- collect constraints
- query the SMT solver for verification targets

Branch Conditions

- auxiliary stack that keeps track of conditions for current point in control-flow
- no constraint added to SMT solver

Control-Flow

- add `b -> r` where `b` is conjunction of branch conditions and `r` is condition in `require(r)` or `assert(r)`

Type Constraints

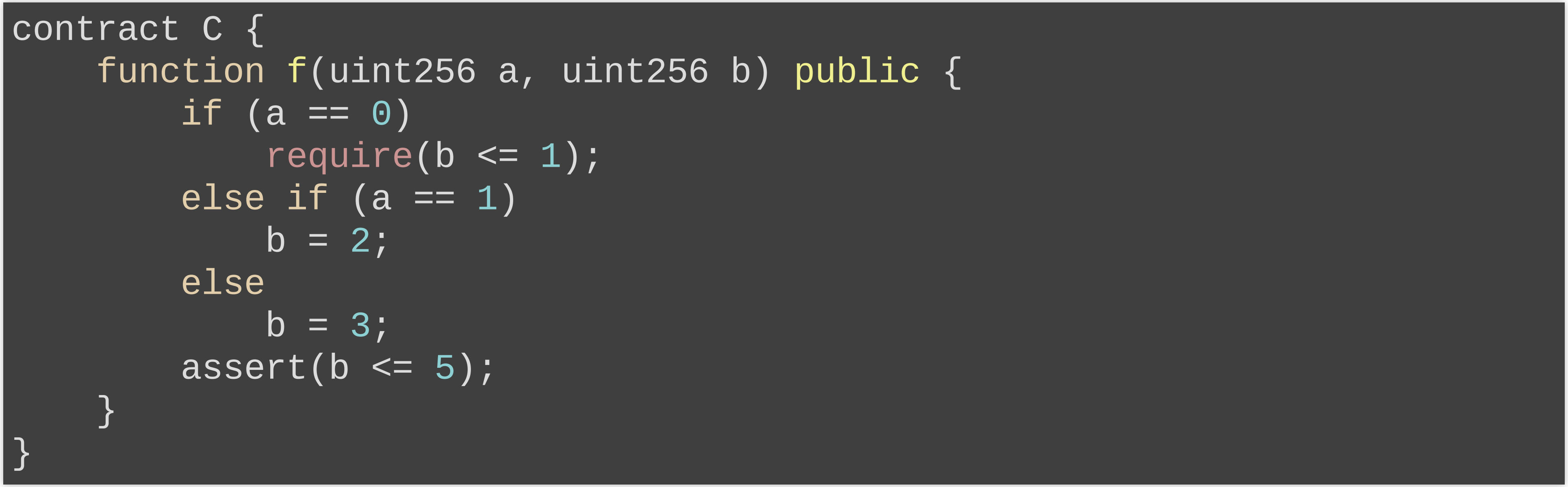
- local variables take default value of type (`0` / `false`), while function parameters take full range of type (`uint`: `0 <= x < 2**256`)

Variable Assignments

- encoding follows SSA form
- control-flow joins use if-then-else function and branch conditions to combine SSA values from different branches

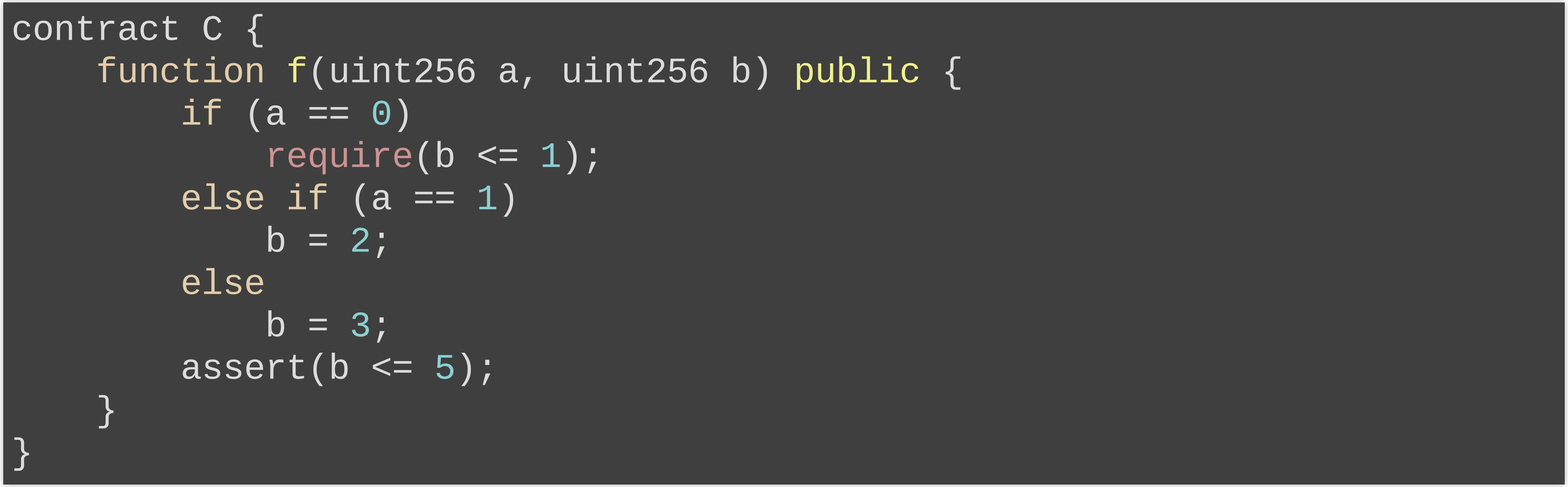
Function Calls

- internal calls fully inlined (might need heuristic at some point)
- external calls reset storage variables to "unknown"



```
contract C {
    function f(uint256 a, uint256 b) public {
        if (a == 0)
            require(b <= 1);
        else if (a == 1)
            b = 2;
        else
            b = 3;
        assert(b <= 5);
    }
}
```

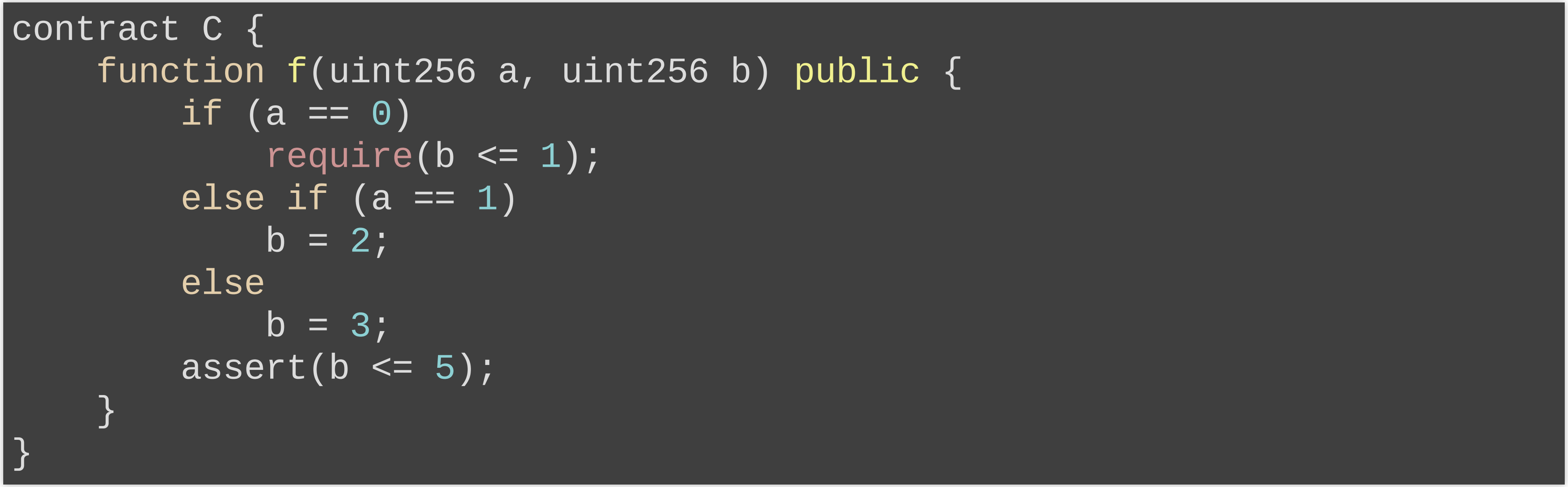
$a_0 \geqslant 0, a_0 < 2^{256},$ $b_0 \geqslant 0, b_0 < 2^{256},$



```
contract C {
    function f(uint256 a, uint256 b) public {
        if (a == 0)
            require(b <= 1);
        else if (a == 1)
            b = 2;
        else
            b = 3;
        assert(b <= 5);
    }
}
```

$a_0 \geqslant 0, a_0 < 2^{256},$ $b_0 \geqslant 0, b_0 < 2^{256},$

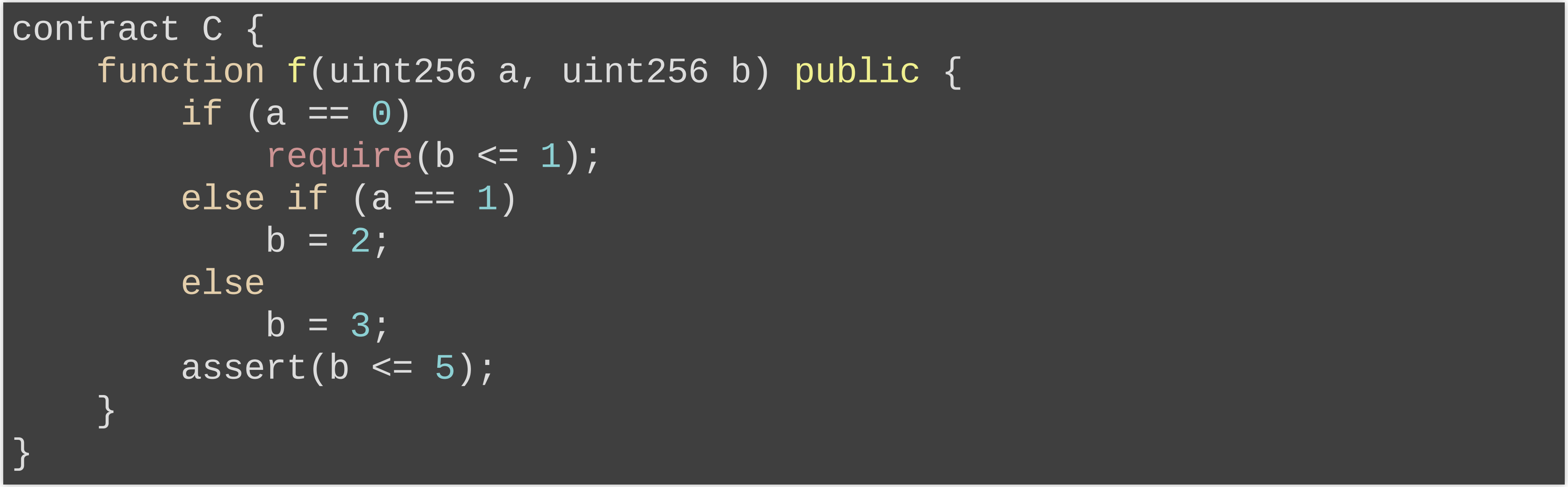
$a_0 = 0 \text{ -> } b_0 \leqslant 1,$



```
contract C {
    function f(uint256 a, uint256 b) public {
        if (a == 0)
            require(b <= 1);
        else if (a == 1)
            b = 2;
        else
            b = 3;
        assert(b <= 5);
    }
}
```

$a_0 \geqslant 0, a_0 < 2^{256},\qquad b_0 \geqslant 0, b_0 < 2^{256},$

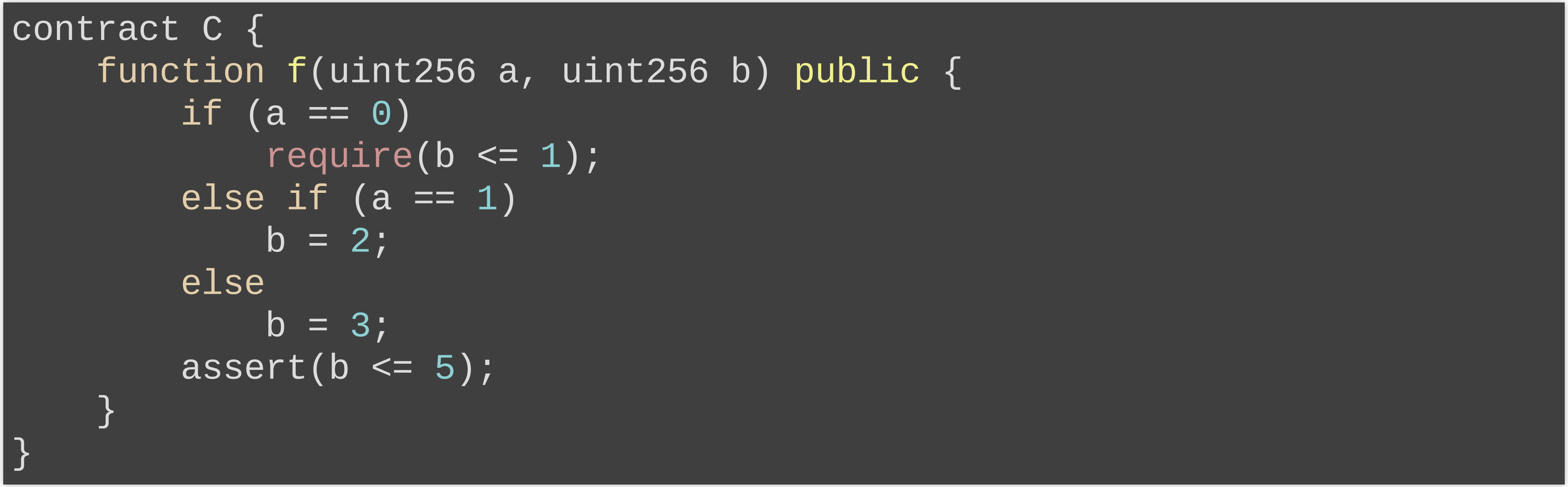
$a_0 = 0 \text{ -> } b_0 \leqslant 1,\quad b_1 = 2,$



```
contract C {
    function f(uint256 a, uint256 b) public {
        if (a == 0)
            require(b <= 1);
        else if (a == 1)
            b = 2;
        else
            b = 3;
        assert(b <= 5);
    }
}
```

$a_0 \geqslant 0, a_0 < 2^{256},$ $b_0 \geqslant 0, b_0 < 2^{256},$

$a_0 = 0 -> b_0 \leqslant 1,$ $b_1 = 2,$ $b_2 = 3,$

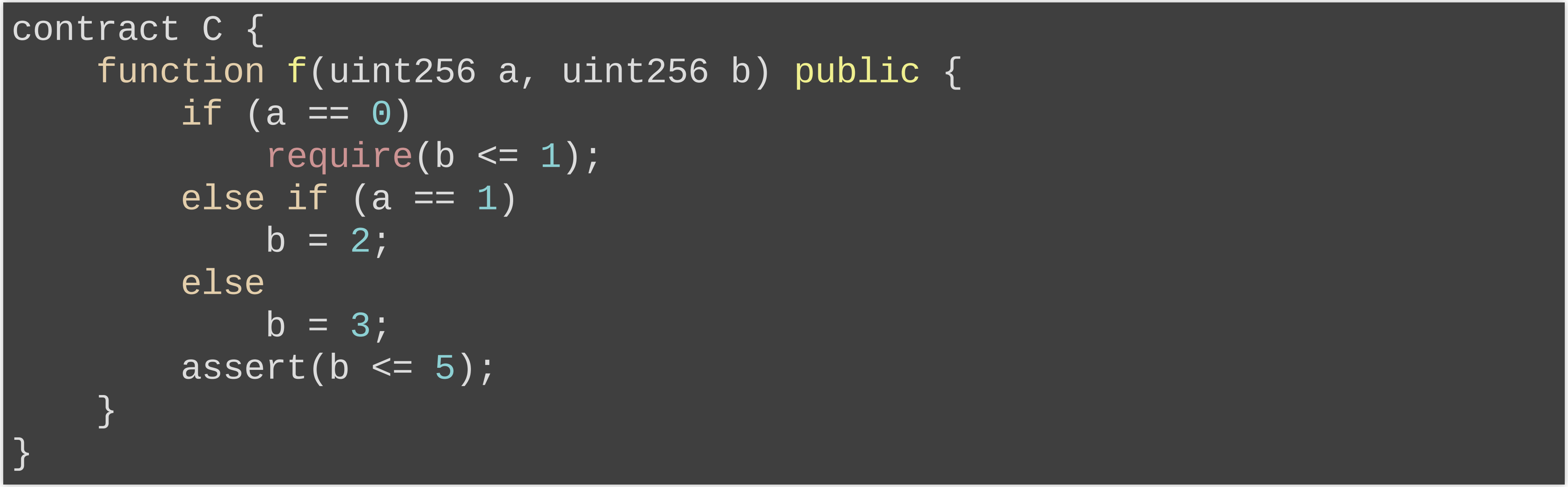


```
contract C {
    function f(uint256 a, uint256 b) public {
        if (a == 0)
            require(b <= 1);
        else if (a == 1)
            b = 2;
        else
            b = 3;
        assert(b <= 5);
    }
}
```

$a_0 \geqslant 0, a_0 < 2^{256}, \quad b_0 \geqslant 0, b_0 < 2^{256},$

$a_0 = 0 \text{ -> } b_0 \leqslant 1, \quad b_1 = 2, \quad b_2 = 3,$

$b_3 = ite(a = 1, b_1, b_2),$

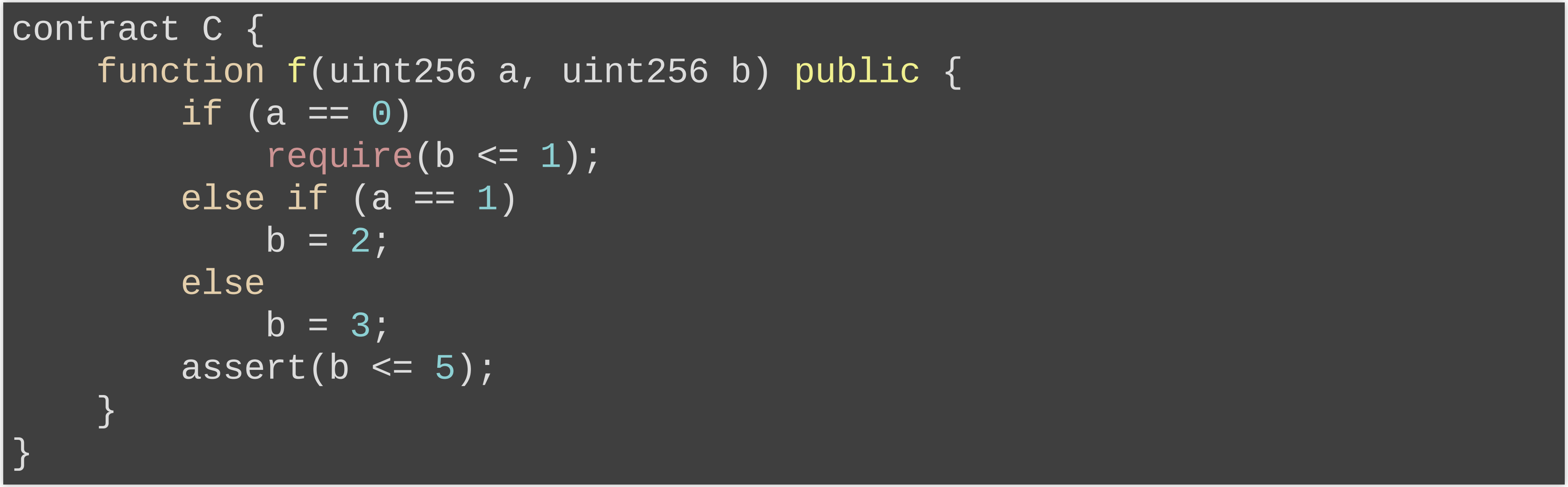


```
contract C {
    function f(uint256 a, uint256 b) public {
        if (a == 0)
            require(b <= 1);
        else if (a == 1)
            b = 2;
        else
            b = 3;
        assert(b <= 5);
    }
}
```

$a_0 \geqslant 0, a_0 < 2^{256}, \qquad b_0 \geqslant 0, b_0 < 2^{256},$

$a_0 = 0 \text{ -> } b_0 \leqslant 1, \quad b_1 = 2, \quad b_2 = 3,$

$b_3 = ite(a = 1, b_1, b_2), \qquad b_4 = ite(a = 0, b_0, b_3),$



```
contract C {
    function f(uint256 a, uint256 b) public {
        if (a == 0)
            require(b <= 1);
        else if (a == 1)
            b = 2;
        else
            b = 3;
        assert(b <= 5);
    }
}
```

$a_0 \geqslant 0, a_0 < 2^{256}, \qquad b_0 \geqslant 0, b_0 < 2^{256},$

$a_0 = 0 \text{ -> } b_0 \leqslant 1, \quad b_1 = 2, \qquad b_2 = 3,$

$b_3 = \mathrm{ite}(a = 1, b_1, b_2), \qquad b_4 = \mathrm{ite}(a = 0, b_0, b_3),$

$\neg\, b_4 \leqslant 5$

Future Plans

- automatic detection of loop bounds
- multi-transaction invariants
- auto-inferred post-constructor invariants

auto-inferred post-constructor invariants

```
contract C {
    uint256 a;
    constructor(uint256 x) public {
        require(x <= 100);
        a = x;
    }
    function f(uint256 y) public view returns (uint) {
        require(y <= 100);
        return a + y;
    }
}
```

State variable a is initialized with value at most 100 and never re-assigned.

Future Plans (2)

- modifiers as pre- and post-conditions plus function abstraction
- explicit contract-level invariant annotations

Future Plans (3)

- effective callback freeness (Grossman et al.)
- range restrictions for "real-life" values like
  - number of transactions, amount of ether, gas, block.timestamp, …

Advanced version migth prove that there is no overflow in the following:

```
contract Coin {
  mapping(address => uint) balances;
  function mint(address r, uint amount) public {
    require(amount < 2**100);
    balances[r] += amount;
  }
  function transfer(address to, uint amount) public {
    require(balances[msg.sender] >= amount);
    balances[msg.sender] -= amount;
    balances[to] += amount;
  }
}
```

Join the discussion!

https://gitter.im/ethereum/solidity-dev

chris@ethereum.org, leo@ethereum.org

We are hiring and giving out research grants!