Formalising the EVM in Dafny

Franck Cassez   Joanne Fuller   Milad K. Ghale
David J. Pearce   Horacio M. A. Quiles

ConsenSys

Ethereum

Ethereum: Overview

> "Ethereum is a decentralized, open-source blockchain with **smart contract** functionality."
> –Wikipedia

- Second largest Cryptocurrency (after Bitcoin)
- A blockchain based on **Proof-of-Stake**
- Allows for "programmable" transactions

Ethereum: Smart Contracts

> "A **smart contract** is a computer program or a transaction protocol that is intended to automatically execute, control or document events and actions according to the terms of a contract or an agreement."
>
> –Wikipedia

- Typically written in **Solidity**
- Other **languages**: *Vyper*, *Rust*, *Fe*
- **Examples**: *multi-signature wallet*, *tokens*, *escrow*, *casino* ...

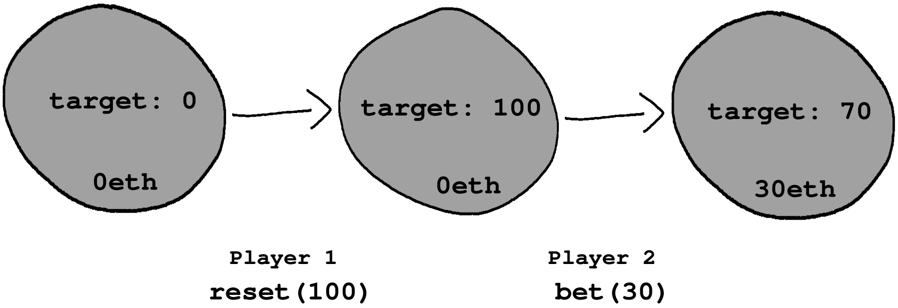
Ethereum: Example Smart Contract

```
contract Betting {
  uint public target = 0;

  function bet() public payable {
    require(msg.value <= target);
    unchecked { target = target - msg.value; }
    if(target == 0) {
      payable(msg.sender).transfer(address(this).balance);
    }
  }

  function reset(uint newTarget) public {
    require(newTarget <= 1 ether);
    require(target == 0);
    target = newTarget;
} }
```

Ethereum: Betting Contract Transition Diagram



Ethereum: Costly Mistakes

The DAO	\$50M	*Reentrancy*
BeautyChain	BEC $\Rightarrow$ \$0	*Integer Overflow*
Akutar NFT	\$34M	*Unreachable Code*
0x	-	*Inline Assembly*
Parity Wallet	\$30M	*Authorisation*
Solana Wormhole Bridge	\$320M	*Signature Check*
Qubit	\$80M	*Logic Error*
MonoX	\$31M	*Logic Error*

Solidity: Deposit Contract

```
deposit(...)
{
  while C1 {
    if C2 return;
    ...
  }
  // As the loop should always end prematurely with the ‘return‘
  // statement, this code should be unreachable. We assert ‘false‘
  // just to be safe.
  assert (false);
}
```

–Cassez, *et al.*, FM’21

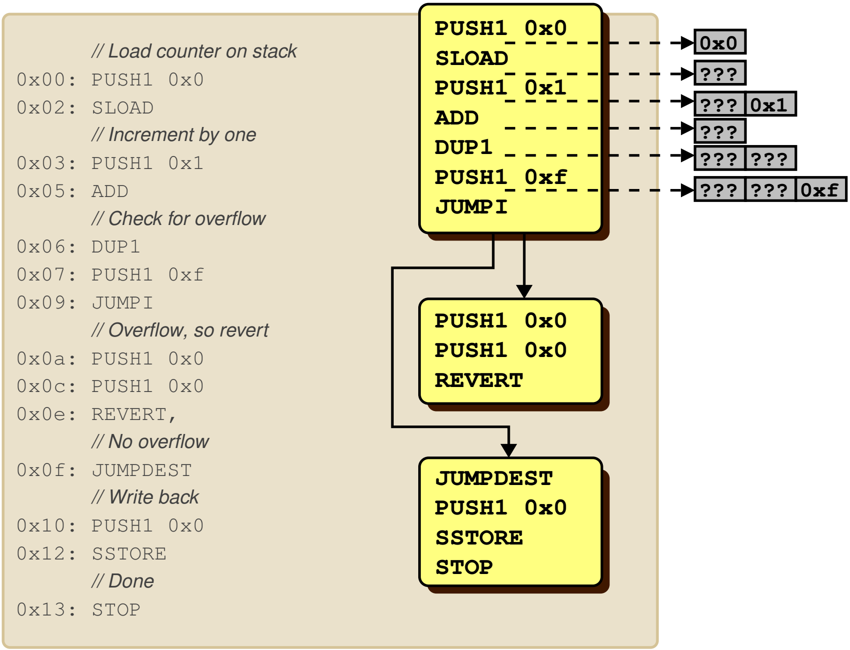
(contract currently holds around 9million ETH)

Ethereum Virtual Machine (EVM)

EVM: Overview

```
PUSH1 0x10
MLOAD
PUSH1 0x20
SLOAD
ADD
PUSH1 0x20
SSTORE
```

- **Stack.** For instruction operands.
- **Memory.** Temporary for contract call.
- **Storage.** Persistent across contract calls.



// Load counter on stack
0x00: PUSH1 0x0
0x02: SLOAD
// Increment by one
0x03: PUSH1 0x1
0x05: ADD
// Check for overflow
0x06: DUP1
0x07: PUSH1 0xf
0x09: JUMPI
// Overflow, so revert
0x0a: PUSH1 0x0
0x0c: PUSH1 0x0
0x0e: REVERT,
// No overflow
0x0f: JUMPDEST
// Write back
0x10: PUSH1 0x0
0x12: SSTORE
// Done
0x13: STOP
PUSH1 0x0
SLOAD
PUSH1 0x1
ADD
DUP1
PUSH1 0xf
JUMPI
0x0
???
??? 0x1
???
??? ???
??? ??? 0xf
PUSH1 0x0
PUSH1 0x0
REVERT
JUMPDEST
PUSH1 0x0
SSTORE
STOP

EVM: The Yellow Paper

Value	Mnemonic	δ	α	Description
0x00	STOP	0	0	Halts execution.
0x01	ADD	2	1	Addition operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] + \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x02	MUL	2	1	Multiplication operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] \times \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x03	SUB	2	1	Subtraction operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \boldsymbol{\mu}_{\mathbf{s}}[0] - \boldsymbol{\mu}_{\mathbf{s}}[1]$
0x04	DIV	2	1	Integer division operation. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv \begin{cases} 0 & \text{if} \quad \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ \lfloor \boldsymbol{\mu}_{\mathbf{s}}[0] \div \boldsymbol{\mu}_{\mathbf{s}}[1] \rfloor & \text{otherwise} \end{cases}$

ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER
BERLIN VERSION beacfbd – 2022-10-24

DR. GAVIN WOOD
FOUNDER, ETHEREUM & PARITY
GAVIN@PARITY.IO

Abstract. The blockchain paradigm when coupled with cryptographically-secured transactions has demonstrated its utility through a number of projects, with Bitcoin being one of the most notable ones. Each such project can be seen as a simple application on a decentralised, but singleton, compute resource. We can call this paradigm a transactional singleton machine with shared-state.

Ethereum implements this paradigm in a generalised manner. Furthermore it provides a plurality of such resources, each with a distinct state and operating code but able to interact through a message-passing framework with others. We discuss its design, implementation issues, the opportunities it provides and the future hurdles we envisage.

1. Introduction

With ubiquitous internet connections in most places of the world, global information transmission has become incredibly cheap. Technology-rooted movements like Bitcoin have demonstrated through the power of the default, consensus mechanisms, and voluntary respect of the social contract, that it is possible to use the internet to make a decentralised value-transfer system that can be shared across the world and virtually free to use. This system can be said to be a very specialised version of a cryptographically secure, transaction-based state machine. Follow-up systems such as Namecoin adapted this original "currency application" of the technology into other applications, albeit rather simplistic ones.

Ethereum is a project which attempts to build the generalised technology; technology on which all transaction-based state machine concepts may be built. Moreover it aims to provide to the end-developer a tightly integrated end-to-end system for building software on a hitherto unexplored compute paradigm in the mainstream: a trustful object messaging compute framework.

1.1. **Driving Factors.** There are many goals of this project; one key goal is to facilitate transactions between consenting individuals who would otherwise have no means to trust one another. This may be due to geographical separation, interfacing difficulty, or perhaps the incompatibility, incompetence, unwillingness, expense, uncertainty, inconvenience, or corruption of existing legal systems. By specifying a state-change system through a rich and unam-

is often lacking, and plain old prejudices are difficult to shake.

Overall, we wish to provide a system such that users can be guaranteed that no matter with which other individuals, systems or organisations they interact, they can do so with absolute confidence in the possible outcomes and how those outcomes might come about.

1.2. **Previous Work.** Buterin [2013a] first proposed the kernel of this work in late November, 2013. Though now evolved in many ways, the key functionality of a blockchain with a Turing-complete language and an effectively unlimited inter-transaction storage capability remains unchanged.

Dwork and Naor [1992] provided the first work into the usage of a cryptographic proof of computational expenditure ("proof-of-work") as a means of transmitting a value signal over the Internet. The value-signal was utilised here as a spam deterrence mechanism rather than any kind of currency, but critically demonstrated the potential for a basic data channel to carry a *strong economic signal*, allowing a receiver to make a physical assertion without having to rely upon *trust*. Back [2002] later produced a system in a similar vein.

The first example of utilising the proof-of-work as a strong economic signal to secure a currency was by Vishnumurthy et al. [2003]. In this instance, the token was used to keep peer-to-peer file trading in check, providing "consumers" with the ability to make micro-payments to "suppliers" for their services. The security model afforded by the proof-of-work was augmented with digital signatures and a ledger in order to ensure that the historical record

EVM: Execution Specs

```
def add(evm: Evm) -> None:
  evm.gas_left = subtract_gas(evm.gas_left, GAS_VERY_LOW)
  x = pop(evm.stack)
  y = pop(evm.stack)
  result = x.wrapping_add(y)
  push(evm.stack, result)
  evm.pc += 1
```

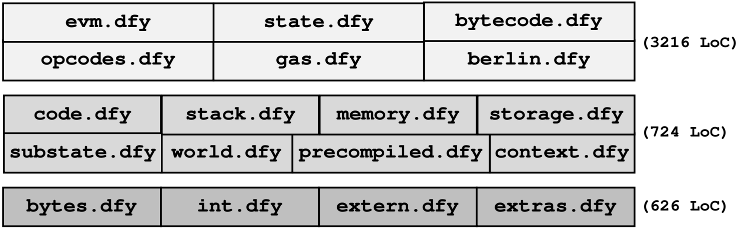
- **Replaces** the Yellow Paper
- Implemented in **Python**
- Can execute against **Common Tests**

EVM: Benefits of Mechanised Formalisation

- **Executable** specification can be validated
- Useful for **sanity checking** new EIPs
- Useful for **verifying** bytecode sequences
- Useful for developing **verified** or **certifying** compilers

DafnyEVM

DafnyEVM: Overview



- **Functionally pure** — no need to specify a specification!
- Bytecode semantics are **state transformers**
- Executable using Dafny backends (currently Java & Go)
- Of 13K common tests, $6900/7500 = 92\%$ **passing**

DafnyEVM: Machine State

```
datatype ExecutingEvm = EVM(
  gas: nat,
  pc: nat,
  stack: Stack,
  code: Code,
  mem: Memory,
  world: WorldState,
  ...
)

datatype State = EXECUTING(evm: ExecutingEvm)
    | REVERTS(gas:nat, data:seq<u8>)
    | RETURNS(gas:nat, data:seq<u8>, ...)
    | INVALID(Error)
    | ...
```

DafnyEVM: Semantics of `ADD`

```
function Add(st: ExecutingState): (st': State)
// Execution either continues or halts with stack underflow
ensures st'.EXECUTING? || st' == INVALID(STACK_UNDERFLOW)
// Execution always continues if at least two stack operands
ensures st'.EXECUTING? <==> st.Operands() >= 2
// Execution reduces stack height by one
ensures st'.EXECUTING? ==> st'.Operands() == st.Operands() - 1
{
    if st.Operands() >= 2
    then
        var lhs := st.Peek(0) as int;
        var rhs := st.Peek(1) as int;
        var res := (lhs + rhs) % TWO_256;
        st.Pop().Pop().Push(res as u256).Next()
    else
        INVALID(STACK_UNDERFLOW)
}
```

DafnyEVM: Semantics of `MLOAD`

```
function method MLoad(st: ExecutingState): (st': State)
// Execution either continues or halts with stack underflow
ensures st'.EXECUTING? || st' == INVALID(STACK_UNDERFLOW)
// Execution always continues if at least one stack operands
ensures st'.EXECUTING? <==> st.Operands() >= 1
// Execution does not affect stack height
ensures st'.EXECUTING? ==> (st'.Operands() == st.Operands())
{
  if st.Operands() >= 1
  then
    var loc := st.Peek(0) as nat;
    // Expand memory as necessary
    var nst := st.Expand(loc,32);
    // Read from expanded state
    nst.Pop().Push(nst.Read(loc)).Next()
  else
    INVALID(STACK_UNDERFLOW)
}
```

DafnyEVM: Memory Invariants

$$M(s,f,l) \equiv \begin{cases} s & \text{if} \quad l = 0 \\ \max(s, \lceil (f+l) \div 32 \rceil) & \text{otherwise} \end{cases}$$

- Memory **expands** on demand
- **Implicit** that length is a multiple of 32bytes

```
function method Expand(mem: T, addr: nat) : (r: T)
ensures (addr + 32) <= |r.contents|
ensures (|r.contents| % 32) == 0 {
  ...
}
```

DafnyEVM: Memory Assumptions

> "... referencing an area of memory at least 32 bytes greater than any previously indexed memory will certainly result in an additional memory usage fee. **Due to this fee it is highly unlikely addresses will ever go above 32-bit bounds**" Gavin Wood, Yellow Paper

0x59	MSIZE	0	1	Get the size of active memory in bytes. $\boldsymbol{\mu}'_{\mathbf{s}}[0] \equiv 32\boldsymbol{\mu}_{\mathrm{i}}$
0x5a	GAS	0	1	Get the amount of available gas, including the

- No limit on maximum size of memory!
- Appears should fit into a 'u256'
- No **exception case** provided for memory overflow

DafnyEVM: Semantics of `SELFDESTRUCT`

```
function method SelfDestruct(st: ExecutingState): (st': State)
ensures st'.RETURNS? || st' == INVALID(STACK_UNDERFLOW) || st' == INVALID(WRITE_PROTECTION_VIOLATED)
ensures st'.RETURNS? <==> st.Operands() >= 1 && st.WritesPermitted()
ensures st'.RETURNS? ==>
  var a := st.evm.context.address;
  a in st'.world.accounts
  && st'.world.accounts[a] == st.evm.world.accounts[a].(balance := 0)
  && a in st'.substate.selfDestruct
ensures st' == INVALID(STACK_UNDERFLOW) <==> st.Operands() < 1
ensures st' == INVALID(WRITE_PROTECTION_VIOLATED) <==> st.Operands() >= 1 && !st.WritesPermitted()
```

> Given such a specification, it is possible to use formal verification techniques to demonstrate that a system design is correct **with respect to** its specification.
>
> –Wikipedia

DafnyEVM: Bytecode Proof

```
method AddBytes(x: u8, y: u8) {
   // Initialise an EVM.
   var st := InitEmpty(gas:=1000);
   // Execute three bytecodes
   st := Push1(x);
   st := Push1(y);
   st := Add();
   // Check top of stack is sum of x and y
   assert st.Peek(0) == (x as u256) + (y as u256);
}
```

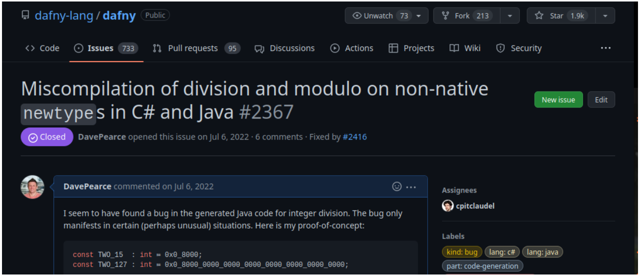
```
method IncProof(st: ExecutingState) returns (st': State)
requires st.PC() == 0 && st.Operands() == 0 && ...
// Success guaranteed if can increment counter
ensures st'.RETURNS? <==> (st.Load(0) as nat) < MAX_U256
// If success, counter incremented by one
ensures st'.RETURNS? ==> st'.Load(0) == (st.Load(0) + 1) {
  var nst := st;
  nst := Push1(nst,0x0);     // Load counter
  nst := SLoad(nst);
  nst := Push1(nst,0x1);     // Increment by one
  nst := Add(nst);
  nst := Dup(nst,1);         // Overflow Check
  nst := Push1(nst,0xf);
  nst := JumpI(nst);
  // Case analysis
  if nst.Peek(0) == 0 {
    assert nst.PC() == 0xa;  // Overflow
    ...
  } else {
    assert nst.PC() == 0xf;  // No overflow
    ...
  }
  return nst;
}
```

DafnyEVM: Practical Experiences

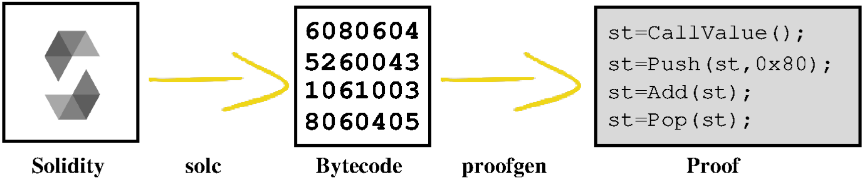
```
AssertAndExpect(() => ReadUint16([0],0) == 0);
AssertAndExpect(() => ReadUint16([0],1) == 0);
AssertAndExpect(() => ReadUint16([0,0],0) == 0);
AssertAndExpect(() => ReadUint16([0,1],0) == 1);
```

- **External Code** — Java calls Dafny *and* Dafny calls Java
- **Continuous Integration** — verified and tested on *every* PR
- **Testing** — want assertions to be verified *and* tested
- `function` or `method` — choose `function method`!

DafnyEVM: Soundness Problems



DafnyEVM: Scaling Up



- ProofGen determines **jump targets** and **stack values**
- ProofGen emits **assertions** for checking overflow/underflow
- Proof needs **manual** tweaking!

```
    assert Memory.Size(st.evm.memory) >= 0x60 && st.Read(0x040) == 0x80; // ADDED BY DJP
    if tmp37 != 0 { block_0x00003d(st); return; }
    st := Dup(st,1);
    st := Push4(st,0xd4b83992);
    st := Eq(st);
    st := Push1(st,0x58);
    var tmp47 := st.Peek(1);
    assume st.IsJumpDest(0x000058);
    st := JumpI(st);
    if tmp47 != 0 { block_0x000058(st); return; }
    block_0x000030(st);
}

method block_0x000030(st': ValidState)
requires st'.PC() == 0x000030
requires st'.Operands() >= 0 && st'.Operands() <= 1
{
    var st := JumpDest(st');
    st := Push1(st,0x00);
    st := Dup(st,1);
    st := Revert(st);
}

method block_0x000035(st': ValidState)
requires st'.PC() == 0x000035
requires st'.Operands() == 1
{
    var st := JumpDest(st');
    st := Push1(st,0x3b);
    st := Push1(st,0x7e);
    assume st.IsJumpDest(0x00007e);
```

DafnyEVM: More Assumptions

```
contract Token {
  mapping(address => uint256) balances;

  ...

  function deposit() {
    balances[msg.sender] += msg.value;
  }
}
```

References

- **Formal and Executable Semantics of the EVM in Dafny**. F. Cassez, J. Fuller, M. Ketabi, D. Pearce and H.M.A.Quiles. In *Proc FM*, 2023. `https://franck44.github.io/publications/papers/dafnyevm-fm-23.pdf`

- **Deductive Verification of Smart Contracts with Dafny**. F. Cassez, J. Fuller and H.M.A.Quiles. In *Proc FMICS*, 2022. `https://arxiv.org/pdf/2208.02920.pdf`

- **Formal Verification of the Ethereum 2.0 Beacon Chain**. F. Cassez, J. Fuller and A. Asgaonkar. In *Proc. TACAS*, 2022. `https://franck44.github.io/publications/papers/eth2-tacas-22.pdf`

http://whiley.org

@WhileyDave
http://github.com/Whiley