A new assembler for the EVM

Ben Siraphob

2020-07-30

Me, briefly

- Undergraduate at Vanderbilt University, TN, majoring in CS and math
- Worked with Ben Scherrey (Proteus Tech, Biggest Fans Productions) on Ethereum smart contracts in the summer of 2019 and 2020
- Interests:
  - Functional and low-level programming
  - Compilers and interpreters
  - Type theory and programming languages
  - Category theory, mathematical logic and algebraic structures

This talk

- Design and implementation of `evm-assembler`
- Example programs and contracts
- Future directions

Quick Facts about Ethereum

- Ethereum is the second largest cryptocurrency platform (by market capitalization)
- Provides a decentralized virtual machine (EVM) to run smart contracts

Quick Facts about EVM

- Instruction set is Turing-complete, within gas limits
- Stack-oriented, 256 byte word size, 1024 items
- Non von Neumann architecture, program stored in virtual ROM
- Memory model is a simple word-addressed byte array
- Exceptional behavior well-defined (e.g. stack underflow, invalid `JUMPDEST`)

Machine state (9.4.1 in Ethereum yellowpaper)

The machine state μ is defined as the tuple (g, pc, m, i, s) which are the gas available, the program counter $pc \in \mathbb{N}_{256}$, the memory contents, the active number of words in memory (counting continuously from position 0), and the stack contents. The memory contents μ_m are a series of zeroes of size 2^{256}.

State of affairs in low-level EVM programming

- Low-level Lisp-like Language (LLL)
  - Macro-oriented (`def`), some control structures (`while`, `if`, `seq`, `for`, `until`) and statements (`set`, `get`, `ref`, `with`)
  - Last docs (0.1 release) in 2017-09-16

State of affairs in low-level EVM programming

- Low-level Lisp-like Language (LLL)
  - Macro-oriented (`def`), some control structures (`while`, `if`, `seq`, `for`, `until`) and statements (`set`, `get`, `ref`, `with`)
  - Last docs (0.1 release) in 2017-09-16
- Ambiguous syntax and semantics, no formalization, incomplete docs

State of affairs in low-level EVM programming

- Low-level Lisp-like Language (LLL)
  - Macro-oriented (`def`), some control structures (`while`, `if`, `seq`, `for`, `until`) and statements (`set`, `get`, `ref`, `with`)
  - Last docs (0.1 release) in 2017-09-16
- Ambiguous syntax and semantics, no formalization, incomplete docs
- Solidity removed their LLL compiler in 2020-01-27

State of affairs in low-level EVM programming

- Low-level Lisp-like Language (LLL)
  - Macro-oriented (`def`), some control structures (`while`, `if`, `seq`, `for`, `until`) and statements (`set`, `get`, `ref`, `with`)
  - Last docs (0.1 release) in 2017-09-16
- Ambiguous syntax and semantics, no formalization, incomplete docs
- Solidity removed their LLL compiler in 2020-01-27
- Vyper uses a custom LLL dialect encoded as Python lists
  - Can be found at `vyper/parser/lll_node.py`

State of affairs in low-level EVM programming

- Low-level Lisp-like Language (LLL)
  - Macro-oriented (`def`), some control structures (`while`, `if`, `seq`, `for`, `until`) and statements (`set`, `get`, `ref`, `with`)
  - Last docs (0.1 release) in 2017-09-16
- Ambiguous syntax and semantics, no formalization, incomplete docs
- Solidity removed their LLL compiler in 2020-01-27
- Vyper uses a custom LLL dialect encoded as Python lists
  - Can be found at `vyper/parser/lll_node.py`

Goal

- A small, correct assembler implementation as a testbed, a suitable intermediate representation for new languages on the EVM.

`evm-assembler` timeline

- Summer 2019
  - First implementation in Scheme, rewrite in Python
- Summer 2020
  - Demonstration of `evm-assembler` contracts calling each other, various demos
  - Ongoing: compilation target for Scherrey's ActorForth language

Example Contract (key-value store)

From the Ethereum development tutorial

```
PUSH1 0 CALLDATALOAD SLOAD ISZERO PUSH1 10
JUMPI STOP JUMPDEST PUSH1 32 CALLDATALOAD PUSH1 0
CALLDATALOAD SSTORE
```

Key-value store contract in evm-assembler

```
100 dup contract-start 1 +    ;; Copy code into memory
0 codecopy 0 return stop
(label contract-start)        ;; Contract start
(org 0)                       ;; Relocate code to address 0
0 calldataload sload iszero   ;; Is key not in store?
(jumpi insert-kv)
stop                          ;; Key already in store, stop
(label insert-kv)             ;; Insert key-value pair
32 calldataload               ;; v := calldataload(32)
0 calldataload                ;; k := calldataload(0)
sstore                        ;; σ'[k] := v
stop
```

Factorial in evm-assembler

```
;; Calculate factorial of 16.
(subroutine factorial)
16 factorial stop

(label factorial) ;; ( n -- n! )
dup 0= fact-base jumpi
dup 1 - factorial *
ret

(label fact-base) ;; ( _ -- 1 )
drop 1
ret
```

Running factorial

```
$ ./run.sh demo/factorial.lisp
Filename: factorial.lisp
Contract size: 104 bytes
Stack: [20922789888000]

Contract code

00000000: 6010 6100 1960 1058 0160 0051 6010 6002  `.a..`.X.`.Q`.`.
00000010: 0a02 0160 0052 565b 005b 8015 6100 5057  ...`.RV[.[..a.PW
00000020: 8060 0190 0361 0019 6010 5801 6000 5160  .`...a..`.X.`.Q`
00000030: 1060 020a 0201 6000 5256 5b02 6000 5180  .`....`.RV[.`.Q.
00000040: 61ff ff16 9060 1060 020a 9004 6000 5256  a....`.`....`.RV
00000050: 5b50 6001 6000 5180 61ff ff16 9060 1060  [P`.`.Q.a....`.`
00000060: 020a 9004 6000 5256                      ....`.RV

Memory

00000000: 0000 0000 0000 0000 0000 0000 0000 0000  ................
00000010: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

Representing instructions

```
@dataclass
class Instruction:
    len: int
    gen: callable // () -> [byte]


def make_inst(len, gen):
    return Instruction(len=len, gen=gen)
```

Assembling simple operations

```
def assemble_simple(a):
    if a in simple_ops:
        return make_inst(len(simple_ops[a]),
                         lambda _: simple_ops[a])
    else:
        raise Exception("Operation not found: {}".format(a))

def assemble_expr(expr):
    if type(expr) == str and expr in simple_ops:
        # expr == simple_op
        return assemble_simple(expr)
    // ...
```

Getting rid of manual PUSH

- `PUSH1` ... `PUSH32` pushes the next 1 ... 32 bytes as a big endian number onto the stack.
- We can write `1312` instead of `PUSH2 1312`, the assembler should generate the appropriate `PUSH`.

Assembling PUSH

```
def assemble_push_then(arg, after):
    if type(arg) == int:
        return make_inst(
            1 + sizeof(arg) + len(after),
            lambda _: [0x5F + sizeof(arg)]
                    + big_endian_rep(arg) + after,
        )
    raise Exception("Invalid operand to push: {}".format(arg))
```

Augmenting EVM with subroutines

- EVM does not have a return stack

Augmenting EVM with subroutines

- EVM does not have a return stack
- Conflates call and parameter stack

Augmenting EVM with subroutines

- EVM does not have a return stack
- Conflates call and parameter stack
- Max contract size is 24 KB $< 2^{16}$

Augmenting EVM with subroutines

- EVM does not have a return stack
- Conflates call and parameter stack
- Max contract size is 24 KB $< 2^{16}$
- Solution: implement call stack by using a single 256-byte word with 16-bit addresses
- Limitation: call depth must not exceed 16

Assembling call

```
def assemble_call(arg):
    if type(arg) == str:
        return make_inst(
            22,
            lambda _: (
                [0x5F + 2]
                + ((lambda push_arg:
                       ([0] if 1 == len(push_arg) else [])
                     + push_arg)
                    (big_endian_rep(resolve_label(arg)))
                )
                + [ *simple_ops["push1"],
                    16,
                    *simple_ops["pc"],
                    *simple_ops["add"],
                    *simple_ops["pushr"],
                    *simple_ops["jump"],
                    *simple_ops["jumpdest"],
                ]
            ),
        )
    else:
        raise Exception("Invalid operand to call: {}".format(arg))
```

Assembling RET

Extra primitives

```
return_stack_loc = 0
forth_words += [
    ("shl", [2, "exp", "mul"]),
    ("shr", [2, "exp", "swap1", "div"]),
    ("pushr", [ return_stack_loc,
                "mload",
                16,
                "shl",
                "add",
                return_stack_loc,
                "mstore",
    ])]
```

Assembling RET (cont.)

Extra primitives (cont.)

```
forth_words += [("popr", [ return_stack_loc,
                           "mload",
                           "dup1",
                           (1 << 16) - 1,
                           "and",
                           "swap1",
                           16,
                           "shr",
                           return_stack_loc,
                           "mstore",
                ], ),
                ("ret", ["popr", "jump"]),
                ("exit", ["popr", "jump"])]

for (x, y) in forth_words:
    simple_ops[x] = assemble_prog(y)
```

Factorial in evm-assembler, revisited

```
;; Calculate factorial of 16.
;; Register factorial as a subroutine
(subroutine factorial)
;; So this becomes 16 (call factorial) stop
16 factorial stop

(label factorial) ;; ( n -- n! )
dup 0= fact-base jumpi
dup 1 - factorial *
ret

(label fact-base) ;; ( _ -- 1 )
drop 1
ret
```

Other program examples in `demo/`

- `factorial.lisp`, `fibonacci.lisp`, `fib-memo.lisp`, `prime.lisp`
  - factorial, fibonacci (recursive), fibonacci (memoized), prime table

Other program examples in demo/

- factorial.lisp, fibonacci.lisp, fib-memo.lisp, prime.lisp
  - factorial, fibonacci (recursive), fibonacci (memoized), prime table
- linked_list.lisp
  - linked lists

Other program examples in demo/

- factorial.lisp, fibonacci.lisp, fib-memo.lisp, prime.lisp
  - factorial, fibonacci (recursive), fibonacci (memoized), prime table
- linked_list.lisp
  - linked lists
- ski.lisp
  - SKI graph reduction VM for lazy λ-calculus

Other program examples in `demo/`

- `factorial.lisp`, `fibonacci.lisp`, `fib-memo.lisp`, `prime.lisp`
  - factorial, fibonacci (recursive), fibonacci (memoized), prime table
- `linked_list.lisp`
  - linked lists
- `ski.lisp`
  - SKI graph reduction VM for lazy λ-calculus
- `assembler.hs`
  - Compile λ-calculus to postfix code for the SKI machine using Oleg Kiselyov's work (λ to SKI, Semantically (2019))

Future directions

- Bitcoin script
- Additional macros
- Target for Scherrey's ActorForth language

Future directions

- Bitcoin script
- Additional macros
- Target for Scherrey's ActorForth language

Goal

- A small, correct assembler implementation as a testbed, a suitable intermediate representation for new languages on the EVM.

Resources

- `evm-assembler`
  - `https://github.com/siraben/evm-assembler`
- Ethereum yellowpaper
  - `https://ethereum.github.io/yellowpaper/paper.pdf`