A modern programming language for smart contracts

Final Report

Nick Roberts[*]
Carnegie Mellon University
Pittsburgh, PA
nroberts@andrew.cmu.edu

ABSTRACT

Blockchain platforms such as Ethereum have risen in popularity with programmers wishing to deploy their own decentralized applications ("smart contracts"). Because execution cost is paid in real currency, it's desirable for smart contract programming languages to provide high-level features that clearly correspond to the cost model of Ethereum.

In this report, we introduce a language for writing smart contracts based on OCaml, a popular multi-paradigm language. The main technical contribution is retargeting the OCaml bytecode compiler to the Ethereum Virtual Machine. We evaluate the performance and usability of this implementation relative to Solidity, the most popular smart contract language. This is a first step towards using Resource Aware ML to automatically predict the execution cost of smart contracts.

KEYWORDS

Compilers, blockchain, Ethereum, Resource Aware ML

1 INTRODUCTION

1.1 Ethereum Smart Contracts

Blockchain platforms serve as a secure, distributed ledger of all transactions within a system wherein the community reaches a consensus on the true sequence of events. Many such platforms support the storage and execution of *smart contracts* as a mechanism for programmers to express state transitions on the system. Of these platforms, Ethereum is the most popular, with over 200 million smart contract transactions executed since 2015.

Transactions in blockchain platforms often involve exchange of quantities of cryptocurrency, and Ethereum is no exception. Each smart contract in Ethereum maintains a balance of *Ether*, which it may send to or receive from other smart contracts and accounts as part of a transaction.

For its execution model, Ethereum uses a stack-based virtual machine called the Ethereum Virtual Machine (EVM). A variety of programming languages targeting this virtual machine have been developed, many of them designed specifically for smart contract development. Several principles guide the design of these languages, emerging from both technical details of the EVM and common applications of smart contracts. We outline some of these guiding principles below:

- **Clear operational semantics.** The cost semantics of the language should have a reasonably clear correspondence to that of the output EVM instructions. This will allow programmers to reason about the performance of their code in the context of the high-level language.
- **Amenability to static analysis.** Extending the previous principle, static analysis tools should be able to analyze the correctness, security, and performance of programs in the high-level language, yielding more insights than the unstructured EVM bytecode.
- **Principled access to primitive operations.** Smart contracts must be able to communicate with other smart contracts, to manipulate the global state by creating accounts and smart contracts, and to exchange Ether with other entities on the blockchain. A smart contract language should expose these capabilities in a way that doesn't compromise safety or correctness of the program.

Technical details of the EVM inform these principles. Foremost, each operation in the EVM incurs a *gas cost*, which is paid in Ether at a rate determined by the transaction. Ultimately, the programmer buys Ether using a real-world currency, so performance of the compiled bytecode matters a great deal, and the cost semantics associated with the high-level language should map clearly onto the actual cost of the bytecode. Furthermore, the underlying operation set of the EVM is Turing-complete, which permits a more expressive frontend language at the expense of complicated static analysis and possible non-termination.

The intent of these principles is to help the programmer avoid costly bugs in both logic and performance, and to prevent adversaries from exploiting vulnerabilities in smart contracts to steal Ether. Existing blockchain languages have not met these goals, as subtleties in semantics have allowed attackers to initiate calls to contracts in inconsistent states [4] or to claim ownership of contracts not belonging to them [2].

1.2 Our Approach

Instead of developing a new language, we target EVM bytecode from an existing language: OCaml, a statically-typed language supporting functional and imperative styles of programming. We implement a library for access to primitives without unsafely exposing internal memory representation.

[*]Undergraduate

2018.

In future work, this implementation could allow for interfacing with existing static analysis tools related to OCaml, such as Resource Aware ML.

The main technical contribution of this paper is a retargeting of the OCaml bytecode compiler to output EVM bytecode. Our implementation supports a rich subset of the OCaml language, including records, arrays, closures, and recursion. We further analyze the performance of the bytecode output by our implementation versus Solidity, the most popular smart contract language, on a set of benchmark contracts.

1.3 Related Work

Solidity is the most popular smart contract programming language, exposing the full Turing-completeness of the EVM with a frontend similar to Java [1]. It lacks some of the features of OCaml, such as first-class functions, and its interface for inter-contract communication has enabled reentrancy attacks such as DAO.

Recent efforts in the formal verification of smart contracts have yielded languages such as Scilla [7], an intermediate-level language that imposes a modal separation between computation and contract communication. Unlike OCaml and Solidity, Scilla is not intended to be a high-level language, for example prohibiting inter-contract calls from appearing in non-tail positions. This limited expressivity makes it a more suitable candidate for an intermediate representation than a frontend language.

Sergey et al. give a full consideration of other smart contract languages suitable for formal verification.

2 CONTRIBUTION

2.1 External Interface of Program

In Ethereum, smart contracts are able to call functions defined in other contracts and use the return value in further computation. For this reason, it is illustrative to introduce a small OCaml program and determine what functions it should make available as a smart contract. Take the following:

```
let compose f g x = f (g x)
let incr x = x + 1
let double x = 2 * x
let f = compose incr double
```

Several points of consideration are immediately apparent.

- Since `compose` is a higher-order function, for it to be available as a callable function to an external client, the client would have to marshall the function argument so that it could be sent over the network.
- The programmer may wish to expose only some of the bindings in the module; for instance, if a function mutates the state of the contract, the programmer may wish to allow only internal calls to it.
- Even though the binding `f` has the same type as `incr` and `double`, the determination of its value requires some computation: in particular, a call to `compose`. When should the value of this binding be calculated, each time an external client makes a call to a function, or upon contract creation?

To address the first two points, we make use of OCaml's module signatures to allow the programmer to specify which functions are available externally. Ethereum's ABI for serializing data makes no provisions for functional arguments, and so we prohibit the module signature from exposing higher-order functions. The following signature is allowable for the module presented earlier; the programmer must not include the specification for `compose`, and chooses not to include the specification for `double`.

```
val incr : int -> int
val f : int -> int
```

To address the third point, we contend that it's desirable for the programmer to decide whether a computation is performed upon contract creation or upon receipt of an external call. For this reason, our compiler allows the programmer to provide two modules: one whose bindings are calculated when the contract is placed on the blockchain (the "setup module"); and another whose bindings are calculated for each external call (the "callable module"). The callable module may reference bindings from the setup module, but not vice-versa.

The EVM does not formally have a notion of function, but Ethereum specifies an ABI for performing function calls between contracts. Natively, the EVM supports only making a call on another contract with some input data. Therefore, the input data necessarily specifies which function the caller wishes to invoke in another contract. As a matter of fact, the first 4 bytes of input data is the first 4 bytes of the Keccak-256 hash of the function signature. Thus, we can intuitively think of the callable module described in this section as exporting a single function `call` with the following behavior:

```
let call input =
  let compose f g x = f (g x) in
  let incr x = x + 1 in
  let double x = 2 * x in
  let f = compose incr double in

  if hd input = hd (keccak256 "incr(int)")
    then incr (tl input)
  else if hd input = hd (keccak256 "f(int)")
    then f (tl input)
  else assert false
```

where `hd` extracts the first 4 bytes of a byte sequence and `tl` extracts all subsequent bytes.

2.2 Extension Point to OCaml Compiler

We extend `ocamlc`, the OCaml bytecode compiler. The output of `ocamlc` is relocatable bytecode object files with the `.cmo` extension, which can be compiled independently (i.e. before all modules are available for linking). The instruction set used in these `cmo` files is similar in spirit to ZINC, the stack-based implementation of ML languages developed in [6]. The execution model of Ethereum is similarly stack-based, making the `cmo` format a suitable input language for our compiler.

The compiler we developed, `evmc`, is a translation of instructions for the Caml Virtual Machine (CamlVM) to instructions for the Ethereum Virtual Machine (EVM). (The semantics of the CamlVM departs from the clean presentation given for ZINC; a full-but-informal specification of the instruction set may be found at [3].) The runtime for EVM bytecode differs from CamlVM bytecode in a few regards:

(i) The CamlVM maintains additional state about execution, including an accumulator value, a pointer to the current variable environment, and a facility to track the number of extra arguments to function application.

(ii) Jump destinations in CamlVM code are reported relative to the currently-executing line of code, but EVM code reports these as absolute offsets from the start of the program.

(iii) The CamlVM allows arbitrary indexing into the stack, but the EVM only allows access to the the top 16 elements.

We address (i) by maintaining the additional information on the EVM heap or as the top element in the stack. The issue noted in (ii) necessitates a more structured intermediate representation with basic blocks that will be explored in the next subsection. Finally, for (iii), we take the same approach as other languages for Ethereum: programs which require access to the stack beyond the 16th element are rejected. This interaction between the the execution model and the high-level language is an unfortunate consequence of the limitations of the EVM.

At this point, we can present an oversimplified description of our compiler: `evmc` translates CamlVM bytecode directly to EVM bytecode. This understanding will be expanded upon as we introduce a more complete picture of the structure of the compiler in later sections.

2.3 Access to Ethereum Primitives

The specification of the EVM includes types not supported in OCaml, such as 256-bit integers, and primitive operations over values of those types. To support these, as the `evmc` compiler translates the `cmo` files, it links calls to these primitive operations against implementations supplied by the compiler. Our implementation provides module signatures for the primitive operations, which supports data abstraction—the `Int256` module signature contains the specifications:

```
type t
val of_int : int -> t
val add : t -> t -> t
val mul : t -> t -> t
...
```

and so the only operations allowable on `Int256.t` are those described in the signature.

The benefit of linking directly against EVM implementations of primitive operations is that function calls can have zero overhead when fully applied. For example, the expression `Int256.(add (of_int 3) (of_int 4))` can be translated directly to the sequence of EVM instructions: `PUSH 4; PUSH 3; ADD`. This efficiency is lost in the face of partial application, however. For the expression `let f = Int256.(add (of_int 3)) in f (of_int 4)`, the compiler must generate a function definition for `add` and partially apply that definition to a single argument, which involves allocating a closure on the heap.

Our compiler also introduces a module for fixed-length byte sequences, named `Etharray`. The signature is as follows:

```
type 'a t
type p (* persistent *)
type h (* heap-allocated *)
val alloc : Int256.t -> h t
val alloc_persistent : Int256.t -> p t
val len : 'a t -> Int256.t
val get : 'a t -> Int256.t -> Int256.t
val set : 'a t -> Int256.t -> Int256.t -> unit
```

The dummy type parameter to `t` allows the compiler to constrain that the setup module contains only bindings to persistently-allocated byte sequences (`p t`). The behavior of `get`, `len`, and `set` is determined at runtime by a flag stored on values of type `'a t` that dictates whether to load from persistent memory or heap memory. In cases where the compiler has enough information to deduce the type argument to `t`, a more efficient implementation is filled in.

This approach for accessing primitive operations emits efficient code while allowing access to the bindings in a type-safe way, in accordance with our listed principles of language design.

2.4 Structure of Compiler

The design decisions made in the previous subsections contribute to the overall shape of the `evmc` compiler, which is developed in this subsection.

The input to the compiler is a series of OCaml source files, designated as either *setup modules* or as *callable modules*, along with their signatures. The output of the compiler is a single file containing the EVM bytecode that, upon execution, runs the setup modules and then copies a program to the blockchain that receives calls from external agents and delegates to the appropriate binding of the callable modules.

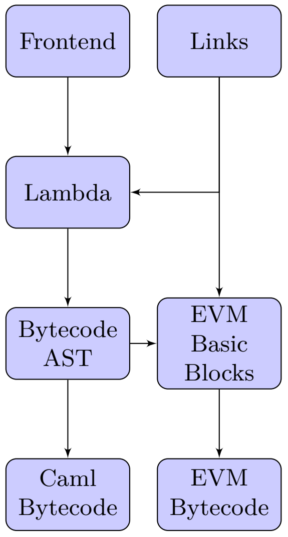

Figure 1: Phases of the `evmc` compiler

Now we expand on the phases of the compiler as shown in Figure 1.

- *Lambda* is a typed IR used in the OCaml compiler. `evmc` inspects the type information available at this phase to determine whether Ethereum protocols can support the user-provided signature for the callable modules. Acceptable function specifications are those of the form `val` $id : \tau_1 \to \tau_2 \to \cdots \to \tau_n$, where each τ_i is one of `Int256.t` or `Etharray.t`.
  Typechecking takes into account the signatures of primitive EVM operations, whose implementations will be linked in a later phase.
- *EVM basic blocks* are translated from an internal representation of Caml bytecode. This is the translation that converts Caml bytecode to EVM bytecode, taking heed of the available instructions as well as the execution model of each.
  The Caml bytecode is inspected to determine when calls to primitive modules are made and the number of arguments provided at call sites. In cases where primitive functions are fully applied, an efficient implementation can be filled in by replacing the function call with the corresponding instruction. Otherwise, a function definition is added to a designated section of the EVM bytecode, and the call site is replaced with a partial application of the new function.
  The callable module is linked against the setup module at this point. The persistent memory addresses of the bindings in the setup module can be statically determined, so these are filled into the callable module at access sites.
  This phase differs from EVM bytecode in that labels, gotos, and function calls are included at a high level of abstraction. This facilitates understanding of the control flow graph of the program and other code analyses.

	OCaml	OCaml (funcs)	Solidity
Code size	1,881	3,334	807
Deployment gas cost	739,158	1,124,557	292,682
`bet` gas cost	74,144	78,832	73,476
`end_lottery` gas cost	15,923	16,330	14,526

Figure 2: Cost metrics of the OCaml contract in Figure 3, an OCaml contract that makes frequent function calls, and a Solidity contract

- *EVM bytecode* is the output of the compiler. Apart from converting labels and gotos to the underlying EVM operations, this phase also performs an optimization pass: peephole optimizations remove unnecessary instructions. The setup modules and callable modules are joined into a single bytecode program, and so the scaffolding to store the callable module on the blockchain is inserted as well.

3 ANALYSIS OF APPROACH

To analyze our translation, we present a simple lottery contract which we evaluate in comparison to an equivalent lottery contract written in Solidity. The full code is given in Figure 3.

The setup module (`setup_lottery.ml`) allocates persistent memory to store the account address of each bettor and the quantity of Ether bet. The callable module (`lottery.ml`) exposes two bindings: `bet`, which bettors call to store their address and their bet in the contract's memory; and `end_lottery`, which the contract creator calls to award the contract's total value to the randomly-selected winner (where the odds of winning are weighted by the amount each participant bet).[1]

3.1 Performance Comparison with Solidity

We evaluate the performance of `evmc`'s translation by writing an equivalent Solidity contract and comparing several aspects of the output code:

- Code size (in bytes).
- Gas cost for deploying the code to the blockchain (including the cost of the code size and the cost of running the setup module).
- Gas cost for calling `bet`.
- Gas cost for calling `end_lottery`.

We evaluate these same metrics on a version of the lottery contract that uses more features of OCaml, including function calls and closure allocation. Figure 2 contains the results.

`evmc` is relatively inefficient in terms of code size; doubly so when using higher-level language features. This explains why the setup cost is so high, since gas is required to store the code of callable module on the blockchain. However, the cost of executing `bet` is similar across the contracts, regardless

[1]The call to `blockhash` is what we use to generate a "random" number, but this is just for purposes of a simple example. The call to `self_destruct` destroys the current contract and sends the remaining balance of the contract to the account specified as an argument.

```
(* setup_lottery.mli *)
val owner : Int256.t
val max_users : Int256.t
val num_users : Int256.t ref
val total_bet : Int256.t ref
val bets : Etharray.p Etharray.t
val users : Etharray.p Etharray.t

(* setup_lottery.ml *)
let owner = P.caller
let max_users = 100
let num_users = ref 0
let total = ref 0
let bets = Etharray.alloc_persistent max_users
let users = Etharray.alloc_persistent max_users

(* lottery.mli *)
val bet : unit → unit
val end_lottery : unit → unit

(* lottery.ml *)
module S = Setup_lottery
module P = Primitives

(* Register a bet for some value of sent Ether *)
let bet () : unit =
  if !S.num_users ≥ S.max_users || P.callvalue = 0
    then P.revert ()
    else begin
      S.total_bet += P.callvalue;
      S.bets.(num) ← P.callvalue;
      S.users.(num) ← P.caller;
      S.num_users += 1;
    end

(* Choose a random bettor to whom to send the contract's
 * total Ether, weighting by the amount bet. *)
let end_lottery () : unit =
  if !S.num_users = 0 || P.caller <> S.owner then () else
    let win = P.blockhash (P.number − 1) % !S.total_bet in
    let rec loop (i : Int256.t) (sum : Int256.t) =
      let sum' = sum + S.bets.(i) in
      if sum' > win then P.selfdestruct S.users.(i)
      else loop (i + 1) sum'
    in loop 0 0
```

Figure 3: Implementation of a simple lottery contract in OCaml.

of language features used. This suggests that the generated code is inefficient only in size, not in the gas incurred by its execution.

3.2 Areas for Improvement

To understand how code size can be reduced, we inspect the result of translating an example function whose execution involves both function application and closure creation. Take the program:

```
let f x =
  let g y z = x + y + z in
  g 3
```

After translation into the Caml Bytecode AST, we obtain the following program, where each instruction is annotated with the corresponding number of EVM instructions output by `evmc`:

```
   RESTART      ;    47 instrs
g: GRAB 1       ;    80 instrs
   ACC 1        ; z   2 instrs
   ACC 0        ; y   2 instrs
   ENVACC 0     ; x   5 instrs
   ADD          ;     1 instr
   ADD          ;     1 instr
   RETURN 2     ;    31 instrs
f: PUSH 3       ;     1 instr
   CLOSURE g    ;    21 instrs
   APPTERM 1, 3 ;    31 instrs
```

The exact details of the Caml instructions can be found in [3], but the semantics can be informally described as follows:

- `RESTART` serves as a target for the resumption of a partially-applied function.
- `GRAB` saves the environment when the function is not fully applied.
- `ACC` accesses arguments to the function.
- `ENVACC` accesses members of the environment.
- `RETURN` and `APPTERM` return to the return address (with the latter performing a tail call to a binding).
- `CLOSURE` closes over the current environment and associates it with a function address that will later be called.

Notably, `GRAB` and `RESTART` only incur their full execution cost if the function is partially applied, but the fact of their large representation as EVM bytecode makes them costly to store as code regardless. Likewise, the cost of executing the `RETURN`, `APPTERM`, and `CLOSURE` instructions are small compared to the cost of storing their code on the blockchain.

This suggests a strategy for reducing the relative cost of using `evmc` over Solidity: aim to output smaller binaries. One strategy for this would be to disallow partial application of functions[2]; this would remove the need for the bloated `GRAB` and `RESTART` instructions.

4 CONCLUSIONS

4.1 Reflection on design principles

In the introduction we identified language design principles that informed our choice of OCaml and our implementation of `evmc`; we now reflect on whether we achieved them.

- **Clear operational semantics.** The translation from the OCaml frontend language to Caml bytecode is based on the ZINC abstract machine, which was created to preserve OCaml semantics at the bytecode level. The next translation is from one stack-based architecture (ZINC) to another (EVM). As such, both

[2]This is the tack taken by Resource Aware ML, though for different reasons.

translations are information-preserving, and a human can reasonably predict the output EVM bytecode based on the source code of the program.

- **Amenability to static analysis.** We discuss this in the subsequent subsection in terms of Resource Aware ML.
- **Principled access to primitive operations.** Our compiler allows the programmer to treat primitive operations as normal OCaml function bindings, but fills in efficient implementations where possible. This balance of expressivity and performance is in line with this principle.

4.2 Future work

4.2.1 Resource Aware ML. The original motivation for this work was to develop a framework for bounding the resource consumption of smart contracts. Resource Aware ML (RAML) [5] is such a resource analysis framework for a language similar to OCaml, so we wished to investigate whether its techniques could be extended to a setting of the blockchain. The utility of RAML is in its ability to derive polynomial upper bounds on the execution cost of functions in terms of their inputs. However, we conjecture that our decision to base our translation on a low-level phase of the OCaml compiler could complicate the application of Resource Aware ML to our implementation. For one, RAML was developed for a structured dynamics, not for a sequence of instructions; and second, the Caml bytecode does not demarcate functions functions, so it's unclear how to identify which sections of bytecode to analyze. We nonetheless propose alternate paths forward:

- *Develop a resource analysis metric on RAML for the EVM.* Developing a metric for RAML is straightforward, but proving any properties about this metric would be difficult, since any analysis would have to survive two translations: one to Caml bytecode (which is unimplemented), and another from Caml bytecode to EVM bytecode (implemented in this paper).
- *Develop a resource analysis metric on Caml bytecode.* Although this has the previously noted difficulties, such a metric would only have to survive one translation, and Caml has a relatively high-level instruction set for application and closure creation. However, since its operation requires type information, RAML would not be a suitable framework for analyzing resource usage in this case.

4.2.2 Compiler optimizations. To make OCaml feasible as a frontend language for Ethereum developers, performance is the path forward. The argument that OCaml provides access to more expressive constructs than Solidity falters in the face of the code size blowup observed in Figure 2. Here, the real-world currency aspect of Ethereum could inform decisions to use another language. For some applications, the code size cost may be acceptable, especially if contracts are called much more often than they're deployed. For applications where contracts are frequently deployed, Solidity is currently the better option.

As noted in the **Areas for Improvement** subsection, the possibility of partial application causes a great deal of code to be generated that is never executed in the phase of full application. Since the OCaml compiler employs separate compilation, it is unable to identify functions that will never be partially applied; with our full-program analysis, we could reduce code size by eliminating the ancillary instructions from appropriate functions.

Minor optimizations could still have far-reaching effects. For example, when the bindings of the callable module are created, an expensive allocation is performed where the bindings exposed in the signature are placed on the heap. This could feasibly be performed entirely on the stack, having implications for both code size and execution cost.

`evmc` achieves its goal of making high-level language features available to blockchain programmers, but performance concerns would push users towards languages like Solidity; addressing these concerns could bring our compiler to see real use.

5 ACKNOWLEDGMENTS

I would like to thank my advisor, Prof. Jan Hoffmann, for support and guidance throughout the research process.

REFERENCES

[1] [n. d.]. Solidity. Retrieved May 10, 2018 from https://solidity.readthedocs.io/en/develop
[2] JD Alois. 2017. Ethereum Parity Hack May Impact ETH 500,000 or $146 Million. Retrieved May 10, 2018 from https://www.crowdfundinsider.com/2017/11/124200-ethereum-parity-hack-may-impact-eth-500000-146-million/
[3] Xavier Clerc. 2010. Caml Virtual Machine. Retrieved May 10, 2018 from http://cadmium.x9c.fr/distrib/caml-formats.pdf
[4] Michael del Castillo. 2016. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft. Retrieved May 10, 2018 from https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/
[5] Jan Hoffmann, Klaus Aehlig, and Martin Hofmann. 2012. Resource Aware ML. In *24rd International Conference on Computer Aided Verification (CAV'12) (Lecture Notes in Computer Science)*, Vol. 7358. Springer, 781–786.
[6] Xavier Leroy. 1990. *The ZINC experiment: an economical implementation of the ML language.* Technical report 117. INRIA.
[7] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR* abs/1801.00687 (2018). arXiv:1801.00687 http://arxiv.org/abs/1801.00687