A Minimal EVM Programming Language for Interactive Verification

Seulkee Baek

3rd October, 2024

Abstract

Blanc is a minimal EVM programming language optimized for formal verification with interactive theorem provers. Its abstraction level sits somewhere between that of Huff and Yul: high enough to avoid the worst pains of writing and verifying programs in EVM bytecode, but low enough to allow easy development and maintenance of associated tools, as well as aggressive contract optimization. The Blanc toolchain is implemented in Lean 4, which includes a verified compiler and formalized semantics of Blanc and EVM. Blanc program verification supports proof of invariant conservation over arbitrary nested executions, including reentrancy. A proof of concept re-implementation of the Wrapped Ether (WETH) contract in Blanc shows encouraging results, including a formal proof of the contract's solvency preservation, a $\approx$3.66$\times$ reduction in bytecode size, and execution cost savings for all functions.

1 Introduction

The case for formal verification of Ethereum Virtual Machine (EVM) smart contracts is well understood. These contracts are small and relatively simple to reason about, at most a few tens of kilobytes in bytecode size. Many of them are mission-critical financial applications whose errors cost up to hundreds of millions of dollars in damage. They are immutable by default and mutability is strongly discouraged due to associated risks, so it is crucial to get everything right at the first attempt. Pesky termination reasoning can usually be waived, thanks to gas limits. All in all, You would be hard pressed to find a software niche with better return on investment for formal verification.

The actual landscape of formal methods for the EVM, however, is not as uniformly well-developed as the high ROI might suggest. Compared to other types of software verification, one feature that immediately stands out is the near-complete absence of interactive theorem proving. There has been interest and research on the subject, and a number of projects are currently in progress; but as far as widely adopted industry tools go, or landmark results on EVM contact verification, very few of them are based on theorem provers like Lean, Coq, or Isabelle/HOL. This is in stark contrast with automated reasoning techniques like SMT solving, symbolic execution, or model checking, which have all found fruitful applications in the field.

This state of affairs is unfortunate, as there are good reasons why automated and interactive proofs generally coexist as complementary approaches. The speed and convenience of automated tools are unbeatable where they work, but relying on them *exclusively* is far from ideal, as they are occasionally hamstrung by a number of well-known issues:

- Little recourse when solver runs out of time/memory.

- Generally weak support for compositional proofs, which are important not only for decomposing a complex proof into manageable sub-problems, but also for efficient reuse of recurring lemmas and human-readability.

- Modern solvers based on sophisticated algorithms incur significant risk of unsoundness from implementation errors. Unfortunately, they also usually do not produce independently verifiable proof objects, so their correctness must be taken on faith.

- Limited expressivity of lower-order languages and proof calculi (which automated systems are typically based on, as a necessary tradeoff for efficient search) precludes many useful proof techniques, such as recursion on user-defined inductive datatypes.

- Difficulty of guiding the solver to perform specific user-chosen inferences at critical steps, which can make dramatic performance differences vs. completely automatic search. Often, the best you can do is to introduce key intermediate results as additional 'lemmas' and count on the solver to figure out the logical connections.

- Solver behavior is not only difficult to control, but at times unpredictable. E.g., an arithmetic constraint may be expressed in two slightly different ways that seem obviously equivalent to a human, but one may completely stump the solver while the other is discharged instantly.

Since all of above are areas that interactive theorem provers *excel* at, it is easy to see the value of having both verification approaches at disposal.

But if interactive verification is so useful, why has it not caught on for EVM smart contracts already, despite years of trying? Although exact reasons must vary by cases, a common thread that runs through previous attempts seems to be that they target *suboptimal abstraction levels*. Some of them go too low, and try to work with EVM bytecode directly: the excessive tedium of doing so practically forces proofs to be almost entirely automated, thereby forgoing key benefits of interactive verification.

Some others aim too high, seeking to verify contracts written in high-level languages like Solidity. The main difficulty of this approach is that, unless it also comes with a verified compiler, proving specifications of high-level programs does not formally imply any guarantees about the bytecode that actually runs on the EVM. And compiler verification for high-level languages is *hard*, on multiple levels: the proof is not only complex, but also requires constant maintenance to keep up with language standard changes and feature additions. Plus, whatever *can* be maintained is often a halfway compromise between performance and verification, with many features disabled in order to make the proof go through. As a result, these approaches based on high-level languages typically only verify their programs against the idealized semantics of the programming language, and take it on faith that the semantics are preserved by an unverified compiler.

Between these low and high extremes, however, there is a relatively unexplored middle ground that is recently beginning to receive more attention. We contend that this intermediate abstraction level offers the ideal tradeoff for interactive contract verification, and present a new language optimized for this purpose. Our specific contributions in this paper are as follows:

- *Blanc*[1], a minimal EVM programming language with an emphasis on control flow graph analysis (Section 2).

- Formalization of Blanc and EVM semantics in Lean 4 (Sections 3 and 4).

[1]`https://github.com/skbaek/blanc/`

- A verified Blanc compiler, with its correctness proof also in Lean 4. (Section 4)
- An inductive method for proving invariant preservation over nested executions, even when it involves calling unknown external contracts and possible reentrancy (Section 5).
- A re-implementation of the Wrapped Ether (WETH) contract in Blanc, with its preservation of solvency over all possible transactions proven at the EVM semantics level, a $\approx 3.66\times$ reduction in bytecode size, and execution cost savings for all functions. (Section 5).

2 The Blanc Language

Generally speaking, the design of a programming language for formal verification is a balancing act between logical rigor and ease of usage. Make it too abstract and high-level, and it becomes hard to establish the logical relationship between source programs and compiled code; make it too simple and low-level, and it becomes difficult to write and verify programs in it. For EVM languages, however, there is a third consideration that tilts the balance: due to the EVM's unusually severe resource constraints and the demand for fine-grained control to navigate around it, as evidenced by strong interest in languages like Huff, there is an additional case to be made from the *user* side for keeping things close to bytecode.

The design of the Blanc language, therefore, aims for minimal abstraction by default, and asks: what is the single most acute pain point for writing verified programs in EVM bytecode? I.e., what is the lightest possible restriction we can impose on raw bytecode that yields the greatest returns in programming ergonomics?

The answer, in our view, is *control flow graph discipline*: for any given piece of code, it should be trivial to immediately identify its sub-components with well-defined entry and exit points. This unlocks compositional programming and verification, where any complex program and its specification can be decomposed into subroutines and their associated lemmas. In addition, it also simplifies programming language semantics by abstracting away jumps and program counters: if we know that subroutine A is followed by subroutine B, we need not worry about implementation details of how/whether that transition is performed with jumps, or what their code locations are.

In order to impose this control flow graph hygiene, we need two things: first, occurrences of *halting instructions* that terminate contract executions (`STOP`, `RETURN`, `REVERT` and `SELFDESTRUCT`) must be limited to designated end-points of a program. Second, arbitrary usage of *jumping instructions* (`JUMP`, `JUMPI`, and `JUMPDEST`) should be disallowed, and replaced with corresponding language constructs. The syntax of Blanc programs is designed to be as simple as possible while enforcing these two restrictions:

```
<function> ::=
    branch <function> <function>
  | last <halting-instruction>
  | next <instruction> <function>
  | call <natural-number>

<program> ::= <function> | <program> , <function>
```

A Blanc *program* is any non-empty list of Blanc functions, whose first item is its 'main' function called upon program execution. A Blanc *function* comes in four different forms[2], each with intuitive semantics:

- `branch f g` pops the top stack item x, and runs `f` if $x = 0$, and runs `g` otherwise.
- `last i` runs a single halting instruction `i`, and terminates.
- `next i f` first runs an instruction `i` (which can be anything other than a halting/jumping instruction), and then runs `f`.
- `call k` runs the (0-indexed) `k`th function of the program. E.g., `call 1` runs the program's first auxiliary function immediately after the main function.

Lean formalization of Blanc functions and programs closely follows the syntax above.

```
inductive Func : Type
  | branch : Func → Func → Func
  | last : Hinst → Func
  | next : Inst → Func → Func
```

[2]The four constructors (**B**ranch, **LA**st, **N**ext, **C**all) are also the namesake of the language.

```
  | call : Nat → Func

structure Prog : Type :=
  (main : Func)
  (aux : List Func)
```

`Hinst` is the type of halting instructions, and `Inst` includes all instructions other than the halting/jumping instructions. This definition ensures that any *successful* execution of a Blanc function must exit via either a `last` or a `call` at one of its endpoints (it could, of course, still fail and revert anywhere mid-program), which allows us to identify intermediate states and the exact sequences of instructions that lead up to them. For instance, if the Blanc function

```
next ADD
next EQ
branch
  last REVERT
  call 3
```

executed successfully, then we may speak of "*the* state *s* after running `ADD` and `EQ` and popping one stack item, and immediately before running the third auxiliary function" because we know such an *s* must exist.

In addition to control flow graph analysis, there are a number of notable features in Blanc's design. The syntax for `call`, for instance, ensures that function calls are always tail calls. This somewhat complicates use of non-tail function calls, which have to be simulated with either inlined code blocks, or more complicated tricks using tail calls and hard-coded return addresses. But this tradeoff is still very much worth it, given how it entirely obviates the call stack and a host of associated edge cases, and also significantly simplifies the compiler and its correctness proof.

Blanc is also unusual in eschewing most of usual language primitives (excluding even loops and variables) and explicitly embracing the stack, a design choice inspired in large part by the Forth [4] language. Although now considered niche for general-purpose computing, successful programming patterns from Forth are arguably more relevant for EVM programming than that of any modern language, given how virtually no other language puts as much emphasis on use of the stack as primary memory and extreme economy for resource-constrained environments. And one of the most enduring lessons from Forth is that, once you have implemented conditionals and function calls/definitions, it is surprisingly easy to build virtually any language feature on top of them and write arbitrary programs in the stack-oriented, point-free style.

This does not mean, of course, that you have to write all Blanc code exclusively *in* the primitives given above. In practice, most Blanc programs will be written with a liberal amount of syntactic sugar. For example, here is the `decimals` function for ERC-20 tokens that returns the number '18', defined as a term of type `Func` in Lean, along with its auxiliary definitions:

```
def Inst.pushWord (w : Word) : Inst := Inst.push ...

abbrev Line : Type := List Inst

def mstoreAt (x : Word) : Line :=
  [Inst.pushWord (x * 32), Inst.mstore]

def pushList : List Word → Line :=
  List.map Inst.pushWord

infixr:65 " ::: " => Func.next

def prepend : Line → Func → Func
  | [], x => x
  | i :: is, x => i ::: prepend is x

infixr:65 " +++ " => prepend

def returnMemoryRange (x y : Word) : Func :=
  pushList [y, x] +++ Func.last Hinst.ret

def decimals : Func :=
  Inst.pushWord 0x12 :::
  mstoreAt 0 +++
  returnMemoryRange 0 32
```

There are a couple notable points here. First, these definitions are (obviously) Lean terms and not written in Blanc's concrete syntax. When Blanc's toolchain matures, we will likely have a two-stage workflow where Blanc code is written in its own syntax with a separate IDE, which will be automatically translated into equivalent Lean terms of type `Prog` and `Func` only when necessary for verification. But it is hard to justify this added complexity at the current nascent state of Blanc and its use cases, so we define Blanc programs directly as Lean terms for now. In addition to being simpler, this also allows us to piggyback on Lean's extensibility for defining new terms and notations.

The second interesting point (and main motivation for the example) is that `decimals`, a very simple function that merely returns a constant number, is defined using no less than *eight* supporting definitions and notations: `pushWord` for pushing a 256-bit word to stack, implicitly converted into a `PUSHN` instruction for an appropriate value of `N`; `Line` as abbreviation for linear blocks of instructions; `mstoreAt n` for storing a 256-bit word at the `n`th position in memory; `pushList` for pushing a list of items to stack; `prepend l f`, which forms a new function by concatenating a line `l` before a function `f`; infix notations for both `next` and `prepend`; `returnMemoryRange` for returning a designated range in memory. This kind of refactoring of common patterns is essential for any realistic contract programming; in practice, programmers will use raw Blanc primitives only when defining the simplest helper terms (like `Func.last Hinst.ret` in `returnMemoryRange`), and most Blanc programming will be done at a higher abstraction level using the defined terms.

3 EVM Semantics

Even when two formal semantics describe the exact same system, their concrete styles can differ widely depending on their objectives. In EVM-related projects, for instance, some formalizations are intended to double as a reference or documentation for the EVM, and much care is taken to make them as unambiguous and informative as possible. Some of them focus on extracting executable Ethereum clients, and invest considerable effort into optimizing the extracted program's performance. The formalization of EVM semantics for Blanc, on the other hand, has no secondary objectives and is purely a pragmatic means for making proofs go through. Its sole guiding principle is to save as much work as possible by maximally *underspecifying* the EVM.

To put it slightly differently: the formalization is merely a *model* of the EVM, not EVM itself. It is very important that everything it *does* say about the EVM is 100% correct, which ensures anything deduced from the formalization is also correct, but not so important that the model includes every minute detail. In fact, for practical purposes it is best to make it contain the *minimum* possible information that still allows you to perform program verification.

For a concrete example, consider the Lean definition of `State`, the type that represents EVM states:

```
structure State : Type :=
  -- balance, storage, & code: parts of the world state
  (bal : Addr → Word)
  (stor : Addr → Word → Word)
  (code : Addr → Bytes)
  -- stack, memory, & return data from last call:
  -- parts of the machine state
  (stk : List Word)
  (mem : Word → Byte)
  (ret : Bytes)
  -- addresses marked for destruction: part of the substate
  (dest : List Addr)
```

`Addr`, `Word`, and `Bytes` are the types of 160-bit addresses, 256-bit words, and byte sequences, respectively. Compared to actual EVM states, a whole host of items are missing from this picture: no nonces in the world state, no remaining gas for contract execution, and the substate almost entirely omitted. Plus, it's also missing some propositional clauses necessary to ensure the state is well-formed, e.g. a proof of `stk.length < 1024` for the stack size limit. Clearly, you cannot define EVM execution as a *function* on this type, as that function's correct behavior would depend on the missing bits.

The trick, however, is that you can get away with much less than a 100% faithful representation by suitably limiting your mission scope. We've already mentioned that Blanc's formal semantics is used only for verification; but we actually go one step further and limit ourselves to verifying only *universal* statements about EVM executions, i.e. *safety* properties. This yields a verification framework that is still quite practical, but also highly tolerant of omissions in its models.

Here's a rough description (with a few missing details, but good enough for an illustration) of how it works: let `State' : Type` be an ideal formalization that, unlike `State` above, captures *all* information in EVM states, and `T' : State' → State' → Prop` be the relation that faithfully represents transition by EVM execution. The goal of EVM program verification, modulo our self-imposed restriction above, is to prove statements of the form

$$\forall\ s_0'\ s_1' : \mathtt{State'},\ T'\ s_0'\ s_1' \to R'\ s_0'\ s_1'$$

Where `R' : State' → State' → Prop` is the predicate of interest that relates the initial and final states of the transition. In other words, we want to show that whenever there is a transition from state s_0' to s_1' by running the EVM, the proposition `R'` s_0' s_1' holds.

Now, the question is: how can we formally prove this this without the hassle of actually formalizing `State'`, `T'`, and `R'` in a theorem prover? For this, we need to formalize three things:

1. A relation `T : State → State → Prop` analogous to `T'`, such that ∀ s_0' s_1', T' s_0' s_1' → T |s_0'| |s_1'| holds, where |·| : `State' → State` is the casting function that removes all parts of `State'` that are not parts of `State`.

2. A relation `R : State → State → Prop` analogous to `R'`, such that ∀ s_0' s_1', R |s_0'| |s_1'| → R' s_0' s_1' holds.

3. Proof of ∀ s_0 s_1 : State, T s_0 s_1 → R s_0 s_1.

Given 1-3, the original goal ∀ s_0' s_1', T' s_0' s_1' → R' s_0' s_1' follows by instantiation and transitivity.

The whole point of this roundabout setup is that finding the suitable `T` and `R` is *far* simpler than a doing a complete and correct formalization of EVM semantics. Condition 2 is especially easy, as its implicit assumption is that `R'` is defined in terms of only the parts that are already in `State` (if not, your definition of `State` is inadequate and should to be extended), so we can simply use `R'` *itself* as `R`, and condition 2 is obviously true. E.g., if `R'` is a description of the EVM stack like "the first state has one more stack item than the second", and it is true for the stripped-down states |s_0'| and |s_1'|, then it is clearly also true for the original states s_0' and s_1'.

Condition 1 is a bit more tricky, but still easy to meet if you exploit the freedom to make the transition relation `T` as lax as needed. Consider, for instance, the execution of precompiled contracts, which tends to be an annoying bottleneck for EVM semantics formalization. The *ideal* way to handle it is to define all the computations in full detail, including complex cryptographic signature, hash, and check algorithms. But it is tricky and time-consuming to get completely right, and frankly a somewhat thankless endeavor if they're not actually used in the contracts you're interested in, and therefore will never appear in the proofs. It's also possible to just put a placeholder function in its place, but this means your formalized semantics now explicitly disagrees with the actual

EVM's behavior, and there's a bit of uneasiness about the logical status of what is being proven.

Once we forego complete formalization as a deterministic function, however, we no longer need to commit to either unpalatable option. In Blanc's EVM formalization, the relation `PreRun` for precompiled contract execution is simply defined as

```
structure Result : Type :=
  (bal : Addr → Word)
  (stor : Addr → Storage)
  (code : Addr → Bytes)
  (ret : Bytes)
  (dest : List Addr)

structure PreRun (s : State) (r : Result) : Prop :=
  (bal : s.bal = r.bal)
  (stor : s.stor = r.stor)
  (code : s.code = r.code)
  (dest : s.dest = r.dest)
```

Where `Result` is a slight variant of `State` that represents terminal states at the end of bytecode executions (hence the omission of stack and memory, which are irrelevant when there is no more code to run). In other words, `PreRun s r` merely states that the EVM world-state and list of addresses marked for destruction are unchanged in a precompiled code execution from `s` to `r`. This is not a particularly informative definition, as it says nothing about the contents of `ret` which is where all the action happens for precompiled contracts. But everything it *does* assert is true and it is easy to check that it is so, which is what matters for ensuring condition 1 above.

To generalize: when choosing the relation `T` to approximate the actual transition relation `T'`, you make `T` *as weak as possible*, by asserting only what you actually need for your verification goals. This is much easier to get right than defining `T'` directly, because it is trivial to verify the small number of assertions in `T`, and whatever `T` omits can't be a source of error. This allows you to remain (safely) ambiguous about irrelevant details, so you can focus on interesting parts that matter for contract verification.

What is irrelevant vs. interesting, of course, depends on your specific goals. This means Blanc's current formalization of EVM semantics is still a work in progress, and it will need to be extended in the future as it is used for specifications on other parts of the EVM. But the general point still stands that this define-as-you-go approach is an effective strategy for parsing the complexity of the EVM in manageable chunks.

Having satisfied that the setup is sound, however, some may still have reservations about limiting program verification to safety properties. The rationale for this restriction is that safety is overwhelmingly more important than liveness for EVM smart contracts, for which nontermination is impossible due to gas limits, and the worst possible outcome of failure is reversion to initial state with wasted gas. We conjecture that the vast majority of risks you want to formally guard against can be stated as safety properties, and a verification tool loses little utility by being restricted to it.

Another somewhat unusual feature in Blanc's EVM semantics is the way it handles nested execution, which needs some clarification. For illustration, consider this failed first attempt at definition of bytecode execution:

```
inductive Exec : Env → State → Nat → Result → Type
  | step :
    ∀ {e s pc s' pc' r},
      Step e s pc s' pc' →
      Exec e s' pc' r →
      Exec e s pc r
  | halt :
    ∀ {e s pc r},
      Halt e s pc r →
      Exec e s pc r
```

`Env` is the type of execution environments, and the `Nat` argument is the program counter at the beginning of execution. The intended meaning of `Exec e s pc r` is "there is a bytecode execution beginning at initial state `s` and program counter `pc` under execution environment `e` that ends at terminal state `r`." This definition, in turn, relies on two auxiliary definitions:

- `Step e s pc s' pc'` means "the execution of a single instruction at position `pc` beginning at initial state `s` under execution environment `e` ends at a new state `s'`, and updates the program counter to `pc'`."

- `Halt e s pc r` means "there is a bytecode execution beginning at initial state `s` and program counter `pc` under execution environment `e` that ends at terminal state `r`, either by (1) executing a single halting instruction at `pc`, or (2) `pc` moving beyond the last instruction of contract code.

In other words, bytecode executions can either end immediately, or execute one instruction and continue. There's nothing wrong with this high-level picture, but the problem is that some of the instructions handled by `Step` (such as `CALL` or `CREATE`) need to perform nested executions themselves. This means the definition of `Step` must use `Exec`, creating a cyclical reference. Although you *could* define them as mutually inductive types, this is really a last resort for when all else fails, as it makes proof by induction on bytecode execution extremely painful to work with.

In the actual definition of `Exec`, we solve this problem by treating the problematic instructions as a special edge case. First, we define a new inductive type `Xinst`, the type of *executing instructions*, and a new relation `Xinst.Run'` which *almost* captures the semantics of executing instructions:

```
inductive Xinst : Type
  | create | call | callcode
  | delcall | create2 | statcall

inductive Xinst.Run' :
    Env → State → Env → State →
    Xinst → Result → State → Prop
  ...
```

The intended meaning of `Xinst.Run' e s ep sp i r s'` is: "running an executing instruction `i` from initial state `s` with execution environment `e` ends in a new state `s'`, *provided* that there is a bytecode execution from `sp` and program counter 0 to `r` under `ep`." This sounds complicated, but it is intuitive if you consider what motivates this definition. Recall that what we *really* want to define here is something like

```
inductive Xinst.Run : Env → State → Xinst → State
    ...
```

Where `Xinst.Run e s i s'` means you can run `i` to get from `s` to `s'` under execution environment `e`. But each constructor of `Xinst.Run` will need to take an argument of type `Exec ep sp 0 r` for some `ep`, `sp`, and `r`, which we cannot use yet. However, this means we *do* know the exact execution environment `ep`, initial state `sp`, and terminal state `r` of the required bytecode execution in each case. Therefore, we define `Xinst.Run'` in the same way we *would* define the hypothetical `Xinst.Run`, except that we omit arguments of type `Exec ep sp 0 r` from each constructor, and expose the types `ep`, `sp`, and `r` in the type signatures instead. This allows us to 'plug in' the missing bytecode execution later, whose conjunction with `Xinst.Run' e s ep sp i r s'` is equivalent to the `Xinst.Run e s i s'` we want.

For a concrete example, see the constructor of `Xinst.Run'` for the `CALL` instruction:

```
| call :
  ∀ (e : Env) exd, e.exd = exd.succ →
  ∀ gas adr clv ilc isz olc osz s stk,
    Stack.Diff [gas, adr, clv, ilc, isz, olc, osz] [1] s.stk
 stk →
  ∀ cld : Bytes, s.mem.Slice ilc isz cld →
  ∀ bal : Balances, Transfer s.bal e.cta clv (toAddr adr) bal
 →
  ∀ r : Result,
  ∀ mem : Memory, storeRet s.mem r.ret osz.toNat olc mem →
    Xinst.Run' e s
      (.prep e s (toAddr adr) cld e.cta clv (toAddr adr) exd
 e.wup)
      (.prep s bal) .call r (.wrap r stk mem)
```

This includes some terms we haven't defined yet, but here's a rough list of what is going on:

- Execution depth `exd` of the nested execution must be 1 less than current execution depth `e.exd` (execution depth starts at 1024 and grows lower in this formalization, which simplifies inductive proofs).
- Final stack stk after execution of `CALL` is obtained by popping 7 stack arguments from the initial stack and pushing a 1.
- Calldata `cld` is taken from initial memory `s.mem`, where its location `ilc` and size `isz` are given as stack arguments.
- ether balance `bal` at the beginning of nested execution is identical to the ether balance at initial state `s`, except for transferring call value amount `clv` from caller address `e.cta` to callee address `toAddr adr`.
- Memory `mem` at terminal state `r` is identical to initial memory `s.mem`, except for overwriting `osz` bytes beginning from location `olc` with `r.ret`, the return data from nested execution.

In other words, there is everything in here for a valid execution of `CALL`—both the preparation of execution environment and state for the new contract execution, and also the unpacking of execution results—except for the actual execution of the called contract's bytecode.

Using `Xinst.Run'`, we can now define `Exec` without any cyclical reference. The definition is identical to the first version, except for addition of the `exec` constructor:

```
inductive Exec : Env → State → Nat → Result → Type
  ...
  | exec :
    ∀ {e s pc ep sp o r s' r'},
      Xinst.At e pc o →
      Xinst.Run' e s ep sp o r s' →
      Exec ep sp 0 r →
      Exec e s' (pc + 1) r' →
      Exec e s pc r'
  ...
```

One caveat is that executing instructions like `CREATE` or `CALL` can actually run *without* a successful bytecode execution, which is an exception not covered by the `exec` constructor above. This is why we define `Step` as:

```
inductive Step : Env → State → Nat → State → Nat → Type
  | reg : ...
  | pre :
    ∀ e s pc ep sp o r s' ,
      Xinst.At e pc o →
      o.isCall →
      Xinst.Run' e s ep sp o r s' →
      PreRun sp r →
      Step e s pc s' (pc + 1)
  | fail :
    ∀ e s pc o s',
      Xinst.At e pc o →
      Fail s o s' →
      Step e s pc s' (pc + 1)
  | jump : ...
  | push : ...
```

Where constructors `pre` and `fail` cover the remaining cases in which an executing instruction runs a precompiled contract, or initiates a bytecode execution that fails. This is also the reason why `Transact`, the relation which describes the execution part of EVM transactions, includes constructors `pre` and `fail` in addition to the main constructors `create` and `call`:

```
structure World := -- type of EVM world states
  (bal : Balances)
  (stor : Storages)
  (code : Codes)

inductive Transact
    (sda : Addr) -- tx sender address
    (rca : Addr) -- tx receiver address
    (w : World)  -- initial world state
    : Result → Prop
  | create : ...
  | call : ...
  | pre :
    ∀ clv bal ret,
      Transfer w.bal sda clv rca bal →
      Transact sda rca w
        { bal := bal, stor := w.stor,
          code := w.code, ret := ret, dest := [] }
  | fail :
    Transact sda rca w {w with ret := .nil, dest := []}
```

And with `Transact`, we can finally define `Transaction`, the type of EVM transactions:

```
structure Transaction (w w' : World) : Type :=
  (vs : Word) -- gas ultimately refunded to sender
  (vv : Word) -- gas ultimately rewarded to validator
  (vb : Word) -- gas ultimately burned
  (nof : vs.toNat + vv.toNat + vb.toNat < 2 ^ 256)
  (sda : Addr) -- tx sender address
  (bal : Balances) -- balances after upfront deduction
  (decr : Decrease sda (vs + vv + vb) w.bal bal)
  (le : vs + vv + vb ≤ w.bal sda)
  (rca : Addr) -- tx receiver address
```

```
    (r : Result) -- execution result
    (act : Transact sda rca {w with bal := bal} r)
    (bal' : Balances) -- balances after refund to sender
    (incr : Increase sda vs r.bal bal')
    (vla : Addr) -- validator address
    (incr' : Increase vla vv bal' w'.bal)
    (del : DeleteCodes r.dest r.code w'.code)
    (stor : w'.stor = r.stor)
```

Note how this definition takes advantage of underspecification like other parts of the formalization: it acknowledges that the quantities `vs`, `vv`, and `vb` exist, but imposes no further conditions on them other than that their sum (1) does not cause a 256-bit overflow, and (2) is small enough to be covered by the sender's ether balance.

Other than underspecification of details and the trick for defining nested executions, Blanc's formalization of EVM semantics is generally straightforward, and can be understood by reading the definitions directly. Currently, `Transaction` is the largest state transition defined because Blanc was only used to verify invariants up to the transaction level, but the formalization can be extended to blocks and related definitions as necessary in the future.

4 Blanc Semantics & Compiler Correctness

Once you have a formalized semantics of the EVM, you can theoretically use it to state and prove any arbitrary statement about EVM contracts. In practice, however, such proofs are likely to be very slow and painful. Suppose, for instance, you want to show that some invariant R holds over a loop. Your informal reasoning is probably something along the lines of:

1. The body of the loop will run n times.
2. Each time the body of the loop runs, it preserves R.
3. By induction on n, the running the loop preserves R.

But if all you have is EVM semantics, this proof becomes an uphill battle from the very first step: how do you even *state* something like 'running the body of the loop' at the EVM level, where there is only a block of unstructured bytecode? If you designate some location as the loop's exit point, how do you show execution must eventually reach it? Given that it must, how do we ensure it always does so by running the loop body *n* times, and never via some completely different path?

Although these questions can all be answered given enough time and effort, this is certainly *not* the level of detail and friction you want to be fighting every step of your way in a proof. That is why, at least for *interactive* verification of EVM contracts, you want to define and work with a higher-level programming language semantics where such questions regarding code paths never arise in the first place.

Fortunately, formalization of Blanc's semantics in Lean is straightforward, as we can repurpose many definitions from the EVM semantics formalization. Recall that in Section 3, we could not define `Xinst.Run`, the intuitive semantics for executing instructions, due to cyclic reference. Now we can, as the relation `Exec` is available:

```
inductive Xinst.Run : Env → State → Xinst → State → Prop
  | exec :
    ∀ {e s ep sp o r s'},
      Xinst.Run' e s ep sp o r s' →
      Exec ep sp 0 r →
      Xinst.Run e s o s'
  | pre :
    ∀ {e s ep sp o r s'},
      o.isCall →
      Xinst.Run' e s ep sp o r s' →
      PreRun sp r →
      Xinst.Run e s o s'
  | fail : ∀ {e s o s'}, Fail s o s' → Xinst.Run e s o s'
```

In addition to being more self-contained than `Xinst.Run'`, this definition has two notable features. First, it includes constructors `pre` and `fail`, thereby covering all possible ways to run an executing instruction. Second, it directly includes the instruction being executed in its type signature, as opposed to using a program counter `pc : Nat` to specify it indirectly. Abstraction of program counters is one of the main differences between formalizations of Blanc and EVM semantics, and part of the reason it is easier to work with the former: it allows you to state and prove theorems about the *thing* that is being executed, instead of reasoning indirectly in terms of its location.

In Section 2, we discussed how all instructions in a Blanc program are either a `Hinst` or `Inst`, where the latter includes all instructions that are not halting or jumping instructions. In Lean, we define `Inst` as:

```
inductive Inst : Type
  | reg : Rinst → Inst
  | exec : Xinst → Inst
  | push : ∀ bs : Bytes, bs.length ≤ 32 → Inst
```

Push instructions get a separate constructor because they are the only 'irregular' instructions with variable bytecode lengths. `Rinst` is the type of all 'regular' `Inst` that is neither a `Xinst`, nor a push instruction. Given the definitions of `Xinst.Run` and `Inst`, we can formalize the execution of `Inst` as:

```
inductive Inst.Run : Env → State → Inst → State → Prop
  | reg :
    ∀ {e s o s'},
      Rinst.Run e s o s' →
      Inst.Run e s (Inst.reg o) s'
  | exec :
    ∀ {e s o s'},
      Xinst.Run e s o s' →
      Inst.Run e s (Inst.exec o) s'
  | push :
    ∀ e {s bs h s'},
      State.Push [Bytes.toBits 32 bs] s s' →
      Inst.Run e s (Inst.push bs h) s'
```

Where `State.Push xs s s'` means the state `s'` is identical to `s` except for pushing items `xs` to the stack (i.e., `xs ++ s.stk = s'.stk`). Using `Inst,Run`, we can now formalize the semantics of Blanc functions that we informally outlined in Section 2:

```
inductive Func.Run :
    List Func → Env → State → Func → Result → Prop
  | zero :
    ∀ {p e s s' f g r},
      State.Pop [0] s s' →
      Func.Run p e s' f r →
      Func.Run p e s (branch f g) r
  | succ :
    ∀ {p e s w s' f g r},
      w ≠ 0 →
      State.Pop [w] s s' →
      Func.Run p e s' g r →
      Func.Run p e s (branch f g) r
  | last :
    ∀ {p e s i r},
      Hinst.Run e s i r →
      Func.Run p e s (last i) r
  | next :
    ∀ {p e s i s' f r},
      Inst.Run e s i s' →
      Func.Run p e s' f r →
      Func.Run p e s (next i f) r
  | call :
    ∀ {p e s k f r},
      List.get? p k = some f →
      Func.Run p e s f r →
      Func.Run p e s (call k) r
```

`State.Pop xs s s'` is the converse of `State.Push xs s s'` (i.e., `s.stk = xs ++ s'.stk`). Note that `Func.Run` takes an argument of type `List Func`, which is necessary as context for defining the `call` case as "calling the `k`th function of the program."

As discussed in Section 2, Blanc program execution is just a special case of function execution that runs the first, or 'main', function of a program.

```
def Prog.Run
    (e : Env) (s : State) (p : Prog) (r : Result) : Prop :=
  Func.Run (p.main :: p.aux) e s p.main r
```

The semantics of Blanc makes verification of Blanc programs much easier than reasoning about EVM bytecode directly. When used in isolation, however, it risks being elaborate deductions about a fictitious system that bears little relation to the code that actually deploys to and runs on EVM. To mitigate this risk, we need to ground Blanc's semantics in that of the EVM with a verified compiler. The Blanc compiler and its correctness proof have many moving parts, but most of it is just tedious bookkeeping to ensure jump addresses match exactly. For our purposes, it suffices to study their type signatures:

```
def Prog.compile (p : Prog) : Option Bytes := ...

theorem correct (e : Env) (s : State) (p : Prog) (r : Result)
  (ex : Exec e s 0 r) (h : some e.ctc = p.compile) :
  Prog.Run e s p r := ...
```

The compiler's return type is an `Option`, as it must account for compilation failures caused by ill-formed Blanc code (e.g., `call` referencing a nonexistent function). In informal terms, `correct e s p r` asserts that (1) if there is an EVM execution from state `s` and program counter 0 to a terminal state `r` under execution environment `e`, and (2) the bytecode of that execution (as specified by `e`) has been compiled from a Blanc program `p`, then there is also an execution of program `p` with the same initial/terminal states and execution environment.

Here's a rough workflow of how this correctness proof is used in program verification. Suppose you want to show that some property holds every time you run the bytecode compiled from a Blanc program `p`. Then we may assume that the *real* proof goal you have in mind has the form

```
∀ (e : Env') (s' : State') (r' : Result'),
  run_program_p e' s' r' → R' e' s' r'
```

Where `Env'`, `State'`, and `Result'` are the 'ideal' versions of `Env`, `State`, and `Result` per the discussion in Section 4, `R' : Env' → State' → Result'` is the property to be verified, and `run_program_p e' s' r'` asserts there is an execution of the contract compiled `p` from `s'` to `r'` under `e'`. Then it suffices, also by reasoning from Section 4, to show

```
∀ (e : Env) (s : State) (r : Result),
  Exec e s 0 r → some e.code = p.compile → R e s r
```

Where `R` is the 'stripped down' analogue of `R'`. Since `Exec e s 0 r` means the current bytecode (as specified by `e`) was executed from position 0, and `some e.code = p.compile` ensures this bytecode was compiled from `p`, their conjunction is equivalent to `run_program_p e' s' r'` above. Now, using theorem `correct`, we can replace this goal with

```
∀ (e : Env) (s : State) (r : Result),
  Prog.Run e s p r → P e s r
```

Which considerably simplifies the proof goal, as you now have a well-behaved premise `Prog.Run e s p r` that you can conveniently decompose into execution of successively smaller parts of `p`.

5 Proof of Concept: Wrapped Ether

Given Blanc's bare-bones features, it is reasonable to question whether it is still a viable programming language. Can you develop real-world smart contracts in Blanc? Did its design concede too much ground to the proof side of things? How does the overall experience compare to existing languages for the EVM?

The best way to answer these questions is to actually write and verify a smart contract with the language. To this end, we re-implemented the Wrapped Ether (WETH) contract in Blanc, and formally verified in Lean at the level of EVM semantics that the contract remains solvent through all possible transactions. Although WETH is a relatively simple token contract, the example is actually quite illustrative in many ways, including some edge case reasoning not generally present for other ERC-20 tokens.

As discussed in Section 2, most Blanc programs are written with a generous amount of extensions with defined terms and notations. For instance, here's the `approve` function from the ported WETH contract in Blanc:

```
-- arguments = [guy, wad]
def approve : Func :=
  arg 0 +++ -- guy ||
  checkNonAddress +++ -- guy_invalid? ||
  .rev <?> -- [if guy is invalid, revert]
  prepApprove +++ -- hash_valid? :: hash :: wad ||
  .rev <?> -- [ if storage location of approval amount
           --   is a valid address that may potentially
           --   collide with balance storage, revert ]
           -- hash :: wad ||
  sstore :: -- ||
  logApprove +++
  returnTrue
```

This example exhibits many features of typical Blanc code. Reusable parts of code are aggressively factored out, even when they're small snippets consisting of a few instructions (`arg n`, `checkNonAddress`, `returnTrue`). Single-use code blocks are also factored out for better organization if they are functionally clearly separated from the rest (`prepApprove`, `logApprove`). Notations are freely defined to suit pragmatic use cases; e.g., `<?>` is the infix version of `branch`, with the branch ordering reversed to reflect a common usage pattern. There's also a liberal amount of highly detailed, line-by-line comments that show updated EVM states after each line and make logical assertions about them. Overall, the 'feel' or writing Blanc code is not unlike that of other low-abstraction EVM languages with extensibility, such as Huff or Yul.

Writing Blanc programs for verification, however, is not merely an exercise in translating Solidity to bytecode with syntactic sugars. Generally speaking, one of the most potent lessons from firsthand verified programming is how programming is much more intertwined with verification than usually imagined. Sometimes, there are small changes to programs that make dramatic differences to verification, and you play a game of global optimization; in other cases, it is revealed during verification that the initial specification is outright *false*, and you are forced to either compromise on the specs or rewrite the program. For re-implementation of the WETH contract, we had to deal with the latter problem, due to the following initial specification:

```
theorem transaction_inv_solvent
    (wa : Addr) (w w' : World)
    (h_code : some (w.code wa) = weth.compile)
    (h_solv : w.Solvent wa)
    (tx : Transaction w w') :
    w'.Solvent wa := ...
```

`wa` is the address of WETH contract, which holds the bytecode compiled from the Blanc program `weth`, according to premise `h_code`. `w` and `w'` are the initial and final world-states of the transaction `tx`. `w.Solvent wa` means the WETH contract at address `wa` is solvent, i.e. that the sum of its liabilities (user WETH balances) is equal to or less than the contract's ether balance. The theorem as a whole states that the WETH contract remains solvent over any transaction.

The statement seems innocent enough, but there are two ways it could go wrong. First, there is a possibility of accounting errors from 256-bit arithmetic overflows: if WETH contract assets > liabilities and adding a deposit amount to both sides makes only the left-hand side overflow, the contract immediately becomes insolvent (we cannot assume assets and liabilities are exactly equal, as the former can be increased at any point by `SELFDESTRUCT`). The chances of this ever happening is practically zero considering how much ether currently exists and its issuance history, but theorem provers have no grasp of common sense, so we need to add an explicit clause that the sum of all ether in existence is less than 2^{256} wei in the initial state.

The second issue is that, if we simply replicate the original WETH contract's algorithm and store user WETH balances and allowance amounts in hashmaps, that data is at risk of corruption from hash collisions. Since users can set allowances to any amount up to $2^{256} - 1$ wei, the contract can go insolvent in a single transaction whenever the storage location for an allowance amount also stores an account's WETH balance. What's worse, we probably won't even know that it happened until that account interacts with the contract, since finding the hash preimage of a given value is intractable.

Unlike the arithmetic overflow issue, however, in this case it is much better to satisfy the original specification by rewriting the contract, rather than adding an additional 'no-collision' clause. Instead using hashmaps, we can simply store the WETH balance of each address x directly to location x of storage, and prevent corruption by explicitly checking and reverting whenever the `approve` function attempts to write to locations $< 2^{160}$ (this is what's happening with the 'revert if valid address' step in the definition of `approve` above). This not only yields a more reliable contract with stronger formal guarantees, but also cuts down bytecode size and gas costs by skipping KECCAK hashing for accessing WETH balances. Low-level programming comes at a cost, but one of its perks is that this kind of optimization naturally suggests itself.

Once the target program and its specification is fixed, verification is mostly proving assertions about intermediate EVM states: given a state $\texttt{s}_0$ for which property $\texttt{P}_0$ holds, you pick the next state $\texttt{s}_1$ reached by running the longest possible line of instructions `l` from $\texttt{s}_0$, and show a new property $\texttt{P}_1$ holds for $\texttt{s}_1$—and repeat until program endpoint. Much of this process is highly repetitive, so we use custom Lean tactics to automate frequently recurring tasks, including:

- Updating the proof goal by 'executing' `l`—i.e., replacing $\texttt{s}_0$ in goal with $\texttt{s}_1$ and adding the hypothesis `Line.Run e` $\texttt{s}_0$ `l` $\texttt{s}_1$ to the context.

- Showing that some some component c of EVM states remains unchanged from $\texttt{s}_0$ to $\texttt{s}_1$. This is easily done with typeclass inference if none of the instructions in `l` alters c.

- Showing that the top stack items of $\texttt{s}_1$ is `xs`, for some `xs : List Word`. This is also simple whenever the stack effect of every instruction in `l` depends only on existing stack items.

The file `Solvent.lean`, which contains all definitions and proofs specific to the verification of WETH contract's solvency preservation (but excludes general-purpose definitions, such as the tactic for showing stack effects), currently consists of ≈1k lines of code. Although this is a feasible number, there is also much room for further improvement with better tooling.

Although the WETH contract's solvency proof is mostly just tedium to be automated away, it does come with one interesting technical twist: when an account unwraps WETH by calling `withdraw`, the unwrapped ether must be sent to caller using the `CALL` instruction. The implicit assumption here is that the call will terminate immediately after the ether transfer; but if the caller is a contract with code, there is nothing that stops it at this point from doing anything a contract can do, including calling the WETH contract again. When your contract must make calls to arbitrary external contracts with potential risk of reentrancy, how do you prove your desired invariant still holds?

Again, we could make a tradeoff here and compromise something on the programming side to mitigate the issue. For instance, we could fine-tune the call gas limit to a bare minimum, which would allow us to formally prove that reentrancy is impossible, but also make the contract brittle against future gas amount changes, or edge case contracts consuming unusual amounts of gas upon receiving ether. Alternatively, we could implement an explicit reentrancy lock, which would be more robust, but also cost more gas every time a user withdraws deposited ether.

Neither solution, however, is truly satisfying. Unlike our previous optimization of WETH balance storage, either fix here yields a strictly inferior and less usable contract. But more importantly, the issue doesn't *feel* like something we should make any concessions to, as it is patently obvious by simple inductive reasoning that our desired invariant holds without any modification. Given the inductive hypothesis that the WETH contract remains solvent over any *nested* executions (including calls to itself), it is easy to show that it remains solvent through an arbitrary bytecode execution. In other words, we don't need to know what the `withdraw` caller does after receiving ether, because we can simply *assume* that it doesn't break solvency, as far as verification goes.

Although the intuition above is sound, it is also somewhat incompatible with our verification setup. Recall from Section 4 that after applying the compiler correctness proof, we get a new proof goal of the following form:

```
(e : Env)
(s : State)
(p : Prog)
(r : Result)
(R : Env → State → Result → Prop)
(h : Prog.Run e s p r)
⊢ R e s r
```

Now, if we were to discharge this goal by induction, we'd have no choice but to induct on `h`, as there is nothing else we can meaningfully induct on. Since Blanc program executions are just a special type of function executions, this induction yields inductive hypotheses about function executions that are 'smaller' or 'lower' than `h`. E.g., if decomposing `h` yields a new proposition `h' : Func.Run fs' e' s' f' r'`, we also get the hypothesis `ih : R e' s' r'` for free. But the problem here is that this is not the inductive hypothesis we actually need: nested executions initiated by `CALL` are *bytecode* executions of the form `Exec e s 0 r`, for which the above induction yields nothing. So a proof by naive structural induction on function execution *after* applying compiler correctness is a dead end.

There are several ways around this problem, but arguably the easiest option is induction on execution depth. Given a Blanc function `f := .next (.exec .call) f'` and its execution `fx : Func.Run fs e s f r`, the nested bytecode execution `cx` initiated by running the first instruction `.exec .call` may not be a direct structural subterm of `fx`, but its execution depth *is* still one level deeper than that of `fx`, so there is a useful sense in which `cx` is 'smaller' than `fx` that makes it eligible for an inductive hypothesis. The exact definitions and proofs involved are a bit complicated, but for our purposes it suffices to examine the key lemma `lift`:

```
def Prog.At (p : Prog) (ca : Addr)
    (e : Env) (s : State) (pc : Nat) : Prop :=
  some (s.code ca) = Prog.compile p ∧
  (e.cta = ca → (some e.code = Prog.compile p ∧ pc = 0))

def ForallSubExec (k : Nat) (ca : Addr) (p : Prog)
    (R : Env → State → Result → Prop) : Prop :=
  ∀ e s pc r,
    Exec e s pc r →
    e.exd < k →
    Prog.At p ca e s pc →
    R e s r

lemma lift
    (R : Env → State → Result → Prop)
    (ca : Addr) -- contract address
    (p : Prog)
    ( depth_ind :
      ∀ {e s r},
        Prog.Run e s p r →
        e.cta = ca →
        ForallSubExec e.exd ca p R →
        R e s r )
    ... :
    ∀ e s pc r,
      Exec e s pc r →
      Prog.At p ca e s pc →
      R e s r := by ...
```

The utility of `lift` is similar to that of `correct` discussed in Section 4—it 'lifts' a messy proof goal about bytecode executions to a new goal `depth_ind` (there are a few others omitted, but this is the main one) where you can work with Blanc program executions and their well-behaved semantics. It uses an auxiliary definition `Prog.At`, where the meaning of `Prog.At p ca e s pc` can be roughly understood as "with the program `p` at address `ca`." This is more complicated than the premise `some e.ctc = Prog.compile p` of `correct`, because we now have to account for the case where the program being executed is not `p`, but their roles are still the same. One crucial difference between `correct` and `lift`, however, is that when using `lift` we get an additional premise `ForallSubExec e.exd ca p R` which asserts that any 'sub-execution' of `e` (i.e., any bytecode execution with depth smaller than that of `e`) satisfies the property `R`.

Using the stronger premises provided by `lift`, we can prove that the ported WETH contract remains solvent over the contract call made by `withdraw` without making any compromises to contract performance. The setup for induction on execution depth is somewhat involved, but the fact it is possible at all is a testament to the flexibility of interactive verification.

In addition to a formal guarantee of solvency, re-implementation in Blanc also pays off handsomely in code density. The deployed bytecode of the ported

	original	port	diff.	% diff.
approve (no change)	4520	4246	-274	-6.062
approve (init. allowance = 0)	24420	24146	-274	-1.122
approve (init. allowance > 0)	7320	7046	-274	-3.743
deposit (deposit amount = 0)	4074	3731	-343	-8.419
deposit (init. balance = 0)	23974	23631	-343	-1.431
deposit (init. balance > 0)	6874	6531	-343	-4.99
withdraw (withdrawal amount = 0)	4440	3883	-557	-12.55
withdraw (withdrawal amount > 0)	13940	13383	-557	-3.996
transfer (transfer amount = 0)	7262	6391	-871	-11.99
transfer (init. receiver balance = 0)	29962	29091	-871	-2.907
transfer (init. receiver balance > 0)	12862	11991	-871	-6.772
transferFrom (tranfer amount = 0)	10148	8816	-1332	-13.13
transferFrom (init. receiver balance = 0)	35648	34316	-1332	-3.737
transferFrom (init. receiver balance >0)	18548	17216	-1332	-7.181
allowance	2717	2300	-417	-15.35
balanceOf	2534	2241	-293	-11.56
decimals	2444	116	-2328	-95.25
name	3110	147	-2963	-95.27
symbol	3264	167	-3097	-94.88
totalSupply	343	215	-128	-37.32

Table 1: Execution costs of WETH functions under various initial conditions. All tests were performed on REMIX IDE with the Cancun VM. Notice that for each function, the gas difference between original and ported contracts remains constant, even though the total gas varies depending on initial conditions.

WETH contract is just 854 bytes, about 3.66 times smaller than that of the original (3124 bytes). Although less dramatic than bytecode size difference, there is also an across-the-board reduction in execution costs as well, as shown in Table 1. Overall, execution of bytecode compiled from Blanc saves a few hundred gas per function call due to better low-level optimization. One conspicuous exception is the >90% reduction in view functions `decimals/name/symbol`, which is likely caused by the Solidity compiler using inefficient encodings for `string` and `uint8` return values.

6 Related Work

EVM smart contracts are attractive formal verification targets for obvious reasons, and there is a great variety of existing techniques and literature on the subject. For our purposes, we limit our discussion to existing work whose approaches are directly related to that of Blanc.

Interactively reasoning about the EVM and its contracts is not new, and has been tried using a variety of theorem provers. This line of work, however, tends to focus on completeness of EVM definition at the expense of practical program verification, often with key challenges unaddressed for the later. For instance, Hirai [3] provides a versatile formalization of EVM semantics which is not only detailed, but can be automatically converted to equivalent definitions in Coq, HOL4, and Isabellle/HOL with Lem's translation feature. For contract verification, however, it needs to assume invariant preservation over reentrancy on the basis of an informal argument, and doesn't offer a framework for compositional reasoning about EVM bytecode, so proofs rely heavily on automated tactics that can take many hours or even longer for each proof goal.

Blanc's support for low-level optimization and arbitrary user-defined specifications is relatively rare for verified EVM programming, but these features have been incorporated before, most notably by a family of solutions that directly verify EVM bytecode with automated reasoning techniques like symbolic execution and SMT solvers. KEVM [2] is a particularly successful example of this group, which has been used in several important results such as verification of the Ethereum 2.0 deposit contract. Compared to Blanc, these approaches offer even greater freedom of programming by accepting arbitrary bytecode, but their proof systems are less flexible than general-purpose theorem provers like Lean.

In terms of overall aim and methodologies, the two projects that Blanc shares the most in common with are Nethermind's Clear [5] and Formal Land's Coq of Solidity [1], which are both based on a relatively low-abstraction EVM language (Yul) and reasons about it with an interactive theorem prover (Lean/-Coq). Since both projects are under development, their details may be subject to future change; but judging by existing code and documentations, their emphasis seems to be (at least initially) on lowering barriers to entry for smart contract developers by providing various convenient interfaces, such as automatic translations that allow users to write high-level programs and specifications. Blanc, in contrast, focuses on maximal flexibility and rigor at the bytecode level, but requires considerable knowledge of both Lean and the EVM for usage.

Finally, the idea of using the WETH contract and its solvency as a proof of concept was taken from a blog post [6] on establishing this property[3] with the Z3 solver. In fact, Blanc began as a weekend project trying to formalize the same results in Lean.

7 Conclusion

We presented Blanc, a EVM programming language optimized for interactive formal verification. Blanc has a minimal amount of syntax which ensures its programs always have simple control flow graphs, but otherwise stays close to EVM bytecode. Its design allows compositional programming and verification, while also making it easy to build and maintain the language toolchain, including a verified compiler. Having a verified compiler allows us to directly state and prove properties of the bytecode that runs on the EVM, instead of working with a high-level language semantics whose connection to deployed bytecode has to be trusted.

Blanc's toolchain is implemented in Lean 4. In addition to the verified compiler, it also includes the formalized semantics of EVM and Blanc that the compiler's correctness proof is based on. Blanc's style of semantics formalization prioritizes soundness of proofs and using minimum implementation effort above all else, at the expense of other desirable traits like completeness or executability. More specifically, its approach to formalization allows safe omission of select parts (e.g., gas calculations or computations in precompiled contracts) such that it affects the scope, but not the soundness, of what can be proven.

[3]The post actually uses 'solvency' to refer to a different property, but still does prove that the sum of the contract's assets is equal to or greater than the sum its liabilities.

We also demonstrated the feasibility of verified EVM programming with Blanc by re-implementing the WETH contract and proving that it remains solvent through all possible transactions. The ported contract showed significant improvements in bytecode size and execution costs, as may be expected from a language that exposes low-level internals to the programmer. But more importantly, verification of Blanc programs was flexible enough to adapt to a new induction technique and show invariant preservation over nested executions that potentially include reentrancy. We consider this a strong evidence of the value of interactive verification for EVM contracts, considering how modelling intercontract calls has been a recurring difficulty often left unaddressed or only informally treated by previous approaches.

References

[1] Coq of solidity – part 1. https://formal.land/blog/2024/06/28/coq-of-solidity-1, Jun 2024.

[2] Everett Hildenbrandt, Manasvi Saxena, Nishant Rodrigues, Xiaoran Zhu, Philip Daian, Dwight Guth, Brandon Moore, Daejun Park, Yi Zhang, Andrei Stefanescu, et al. Kevm: A complete formal semantics of the ethereum virtual machine. In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 204–217. IEEE, 2018.

[3] Yoichi Hirai. Defining the ethereum virtual machine for interactive theorem provers. In *Financial Cryptography and Data Security: FC 2017 International Workshops, WAHC, BITCOIN, VOTING, WTSC, and TA, Sliema, Malta, April 7, 2017, Revised Selected Papers 21*, pages 520–535. Springer, 2017.

[4] Mahlon G Kelly and Nicholas Spies. *FORTH: a text and reference*. Prentice-Hall, Inc., 1986.

[5] František Silváši and Julian Sutherland. Clear–prove anything about your solidity smart contracts. https://medium.com/nethermind-eth/clear-prove-anything-about-your-solidity-smart-contracts-04c6c7381402, Jul 2024.

[6] Luna Tong. Formally verifying the world's most popular smart contract. https://www.zellic.io/blog/formal-verification-weth/, Nov 2022.