Verified Compiler from a Subset of C to Ethereum Bytecode

Lee Danilek
Advised by Zhong Shao
Informally Advised by Vilhelm Sjöberg

May 3, 2018

1 Abstract

Professor Shao's startup Certik will verify Ethereum smart contracts by mathematically proving their properties in the language Coq (Shao). Contract authors will write in the high-level language DeepSEA, which is converted to Coq so the user can write proofs about its functionality. Meanwhile, the DeepSEA is compiled to Ethereum bytecode by a verified compiler that ensures the high-level functionality is correctly executed when the contract runs on the Ethereum blockchain.

My project creates the bottom of the verified compiler: compile a C-like language (MiniC) down to Ethereum bytecode. Do this by writing a compiler in Coq that compiles MiniC through several intermediate languages until finally producing bytecode. The produced bytecode has been tested with a Python emulator of the Ethereum Virtual Machine.

The compilation is split into many steps, each compiling one intermediate language to the next. Here is a list of the intermediate languages:

- Clike - expands data structure operations into pointer arithmetic and hashing.
- Cgraph - turns an abstract syntax tree into a control flow graph of compound instructions.
- Cbasic - expands compound instructions into simple instructions.
- Clinear - flattens the control flow graph into a sequence of instructions.
- Clabeled - re-assigns all labels to make them globally unique, so jump instructions are well defined.
- Stacked - implements function calls while converting local environments to a single stack of values.
- EVM - concatenates everything into a list of bytecode instructions.
- Printer - outputs the bytecode in assembly-like code or as actual executable bytes.

The first five of these compilation steps have proofs of correctness in Coq, which form the bulk of my contribution. For each language, the semantics allow for inter-contract communication through method calls and transfers of ether, and they also quantify the amount of gas required to execute the code.

2 Background and Context

Blockchain is all the rage at the moment, with Ethereum smart contracts making it possible for anonymous individuals to execute transactions safely and securely. The documentation for Solidity, a language for writing smart contracts, highlights several use cases.

For example, a smart contract may implement an auction system, where a seller posts a contract with a specified beneficiary. Then bidders submit increasing bids to the system until the bidding period concludes. The blockchain provides a consensus on the log of bids and the contract doles out the final values to the seller while reimbursing the losing bidders.

As another example, people who wish to perform many transactions quickly can deposit some money in a Payment Channel smart contract. Then they can transact independently with signed messages, using the blockchain for dispute resolution if necessary.

These ideas sound great on the surface, but it's difficult to ensure that the smart contracts are actually working as intended. Bugs in smart contracts occur often, and these bugs lose people real money. By the decentralized anonymous nature of the blockchain, these lost funds are typically lost forever with no hope for recourse.

Contract authors want ways to verify their contracts are mathematically sound. For example, they want to ensure that the amount refunded to a losing bidder in an auction is exactly the amount owed to that bidder. At the same time, users of the contract want validation that they aren't being cheated by a subtly imperfect contract. To this end we want to prove that contracts comply with their specifications. We want to write this proof in the language Coq, so that its correctness is derived using lambda calculus and intuitionist proof techniques, and the Coq compiler automatically verifies the proof.

With this goal in mind, for his project Christopher Fu wrote the correctness proof for the payment channel contract, and Valerie Chen proved some properties about the auction contract.

3 Overall Goal

My project focuses on a practical problem concerning these correctness proofs: the proofs have to reason about the functionality of the contract *as it executes* on the Ethereum virtual machine (EVM). It's not enough to simply write the contract function in Coq and prove its correctness, because the EVM does not run Coq code. The EVM instead runs a domain-specific language which I refer to as "Ethereum bytecode." Later I will discuss this language in-depth, but for now it suffices to say that no one would want to prove contract properties by reasoning about the Ethereum bytecode.

We would like contract authors to write their contracts in the language DeepSEA. This language is concise yet expressive enough to allow most constructs that authors already expect from using Solidity (the Javascript-like language used to write contracts in production today). Even better, DeepSEA compiles to Coq so that the authors may prove high-level properties of the contracts. In order to execute its code, DeepSEA compiles to a C-like language called Clight, which then is compiled by a verified compiler Compcert to assembly code.

My project entails rewriting the bottom part of this pipeline. Essentially my goal was to create a "Compcert for Ethereum." We now can compile DeepSEA to Ethereum bytecode, and many of these compilation steps are proved to be correct. The rest have yet to be proved correct, but have been tested by manual inspection of the produced bytecode and execution on an EVM emulator.

Concretely, I modified Clight in a few EVM-specific ways, such as adding expressions to query transaction values, and differentiating between internal function calls and inter-contract method calls. Then I wrote several intermediate languages below Clight, which are compiled to each other in sequence until finally printing bytecode. Each of these languages comes with semantics, which describe in Coq what the instructions do. Each compilation step is verified by a proof that claims the languages do the same things. That is, if the semantics for the source language say the EVM state changes in a certain way, then the semantics for the compiled code in the destination language say the EVM state changes in the same way.

4 Source and Target Languages

4.1 Source: MiniC

For the first task in the project, I changed Clight into a new language MiniC to add Ethereum-specific features and remove unnecessary features. Clight contained pointer types, function pointers, floating point numbers, labels, gotos, switch statements, builtin statements, type-casting expressions, and continue statements. The compiler from DeepSEA never even generated most of these constructs, and the rest were deemed not useful in the Ethereum environment.

After stripping out these language features, there were several that needed adding. In place of pointer dereferences, MiniC has more constrained expressions for array indexing and hash-table indexing. To replace builtin statements, MiniC has builtin expressions to query contract and transaction state. For example, the following would be a common expression used in contracts:

```
Ehashderef (Evar bid_values) (Ecall0 Bcaller)
```

This expression, written in MiniC syntax of Coq (simplified by excluding extraneous type arguments), refers to a global contract variable called `bid_values` which is a hash table. From within this hash table it finds the bid value associated with the transaction's sender, specified by `Bcaller`.

The modified language MiniC is designed to simplify the original Clight as much as possible while still allowing contract authors to use things they commonly use in Solidity. Hash tables are built-in, as are fixed-size arrays and structs. When an lvalue like `Evar bid_values` is used as an rvalue, it is automatically dereferenced. Expressions can retrieve contract values such as the transaction's caller, but expressions cannot have side-effects.

4.2 Target: Ethereum Bytecode

The target language is similar in spirit to x86 assembly, yet uses many EVM-specific concepts. For example, instead of storing computational values in registers, it makes use of a stack by popping instruction arguments and pushing their results. So while a compiler for assembly may use pseudoregisters to store arguments and move them to the correct registers before calling arithmetic instructions, a stack machine like the EVM can compute expressions without such rearranging. For example, it could compute $(5 + 9)/(8 - 1)$ by pushing the arguments 1 and 8, running the command SUB, then pushing 9 and 5 before running ADD, DIV.

In addition to the stack, the EVM has transient Memory for operations on arbitrary numbers of arguments, such as hashing. For persistent storage, the EVM has a map from words to words called Storage. Note that in the context of the EVM, a "word" is a 256-bit integer.

When executing bytecode, each instruction consumes "gas," a unit of computational resource which has monetary expense. Contract authors want to reason about how much gas they need, to ensure their code will execute to completion before it runs out of gas.

5 Intermediate Languages

Each language in the compilation pipeline is defined up until the function level. A language may share some data structures with others, for example the MiniC expression data structure is used in the first six languages, while its statement data structure is used in the first two. Above the level of the function the languages have a uniform type of "global environment," which consists of named functions which can be called internally, methods which are functions that can be called externally, and the constructor which is called once. This global environment is the main unit that is compiled, because it contains the list of global variables to allocate, and on which the semantics act, because it allows functions to search for other callable functions.

5.1 Clike

We accidentally named this language by frequently discussed the design for a new intermediate "C-like" language. In attempting to compile MiniC directly to Ethereum bytecode, I determined that an intermediate language would help, to expand the implicitly-dereferenced pointer values associated with `Evar` (global variable), `Ehashderef` (hash table entry), `Efield` (struct entry), and `Earrayderef` (array entry).

By analyzing the bytecode for several Solidity contracts, compiled with the official compiler solc, I discovered that solc implements hash tables in a startling way. Instead of storing values in a restricted space and dealing with hash collisions somehow, it leverages the Storage component of the EVM, which is a word-indexed set of words. To store hash table values, it hashes the key together with the hash table's identifier, using the cryptographically secure hash function Keccak-256, and uses this hash to directly index into Storage. This is prone to hash collisions, even across different hash tables and with other contract state variables. We considered implementing our own hash tables to deal with collisions, but decided against it for the following reasons. Firstly, although hash collisions must occur, for a cryptographically secure hash function we assume that hash collisions cannot be found. This is already implicit in the assumption that message signatures and the blockchain itself are secure, so it doesn't hurt to extend this assumption to the actual implementation of contracts. Secondly, our final generated code should be able to compete with solc, not just in verified validity but in speed and economy of bytecode. Inlining the hash table collision-detection would slow down our code and increase its size compared to the existing contract code.

The language Clike does away with data structure constructs like hash tables and brings back pointers. It includes explicit dereferences and computes the necessary pointer arithmetic based on the types of each value. For example, when used as an rvalue, the example (Ehashderef of bid_values on index Bcaller) will compile to the following Clike code:

```
Ederef (Ecall2 Bsha_2 (Evar bid_values) (Ecall0 Bcaller))
```

Note that the dereference, required by using the expression as an rvalue, is now explicit. Furthermore, the hash-table dereference has been compiled to a SHA3 hash of two values: the `bid_values` identifier and the key `Bcaller`.

Much of the complexity of this compilation stage arises from array dereferences and struct field accessors, because they require looking at the types of each element in order to determine the correct offset to use in the pointer arithmetic.

5.2 Cgraph

Note that Clight, MiniC, and Clike all store their statements in an abstract syntax tree. For example, an `Ssequence` contains two sub-statements to execute in succession, and an `Sloop` contains a single statement to infinitely loop until hitting an `Sbreak`.

This abstract syntax tree is not particularly difficult to compile directly to bytecode; in fact for several weeks that was the initial plan. However, validating correctness of this compilation step proved an insurmountable task. For provability, this compilation step has been split into many smaller steps that compile through simpler languages. Some of these languages follow in the footsteps of Compcert to enable proof-technique reuse and some minor code reuse.

The language Cgraph is similar to Compcert's language RTL. Its defining feature is translation from the abstract syntax tree of Clike into the implied control flow graph. In Cgraph, statements don't contain other statements; instead each statement points to its successor or successors. So the end of the statement which was inside an `Sloop` points to the statement that was at the beginning of the `Sloop`. An `Sifthenelse` in Clike compiles to a conditional statement `Scond` which points to two successors: an "if true" statement and an "if false" statement.

The language differs from RTL in that it doesn't mess with pseudoregisters. As described above, pseudoregisters do not correspond to anything useful in the context of a stack machine.

Initially it appeared that making the abstract syntax tree into a graph was unnecessarily jumbling the statements. It takes a tree with a well-defined and meaningful order and generates a graph with no particular order. However, when writing the semantics and proof of correctness, it became clear that making the tree into a graph was the simplest way to quickly deconstruct the tree into manageable pieces.

5.3 Cbasic

Similar to the language LTL in Compcert, Cbasic expands the single statements in the control flow graph into "basic blocks." A basic block is a list of small statements which are always executed in sequence. This compilation phase does not join any pieces of the graph together, even if certain nodes in the graph can only happen in a certain sequence; instead, it expands a composite statement like so:

$$\texttt{Slog exp n} \Longrightarrow \texttt{Slog exp :: Sjump n :: nil}$$

The only difference between Cgraph and Cbasic is that while Cgraph statements contain pointers to their successors, a single typical node in the Cbasic control flow graph is a list of two statements, one to actually execute the statement and the other to explicitly `Sjump` to the next node.

This compilation step is small in terms of the changes to the code, but the changes in the semantics are large. To avoid hellishly long proofs in the linearization, this step expands semantics by allowing a "partial execution of statement" state, which allows us to reason about the jumping between statements separately from the actual execution of the statements.

5.4 Clinear

The next language takes cues from Compcert's language Linear as it lays out all of the code for each function in a single list. Clinear flattens the control flow graph from Cbasic, inserting an `Slabel` before each node. The Compcert linearization implements several heuristics to find a total order on the instructions that minimizes the number of jumps necessary. For the initial implementation of the Ethereum version, we stick basic blocks together in an arbitrary order, because order doesn't functionally matter and jumps are fairly cheap operations.

This sequence of simple statements begins to look like the goal bytecode, because it's in the same order and has most of the same instructions. However, there are some compilation steps left before we can output the goal.

5.5 Clabeled

The language Clabeled is required because the labels in the Clinear code are exactly the identifiers for nodes in the control flow graph. These identifiers are unique within functions by design, but they are shared between functions and don't leave any labels for function entry-points or function call return labels.

To compile from Clinear to Clabeled the code first constructs a `label_map`, which is a data structure containing a map from the nodes in each function to labels, along with the next label to assign. Because this is Coq, the data structure also contains a proof that the next label is larger than than any assigned previously, and a proof that the map is injective. Once this label map is constructed, it defines a way to uniquely map each label existing in Clinear code to a globally unique label.

In addition to reassigning all existing labels, this compilation pass also puts labels at the beginnings of functions, followed by an `Sintro` command to execute the function's required preamble.

5.6 Stacked

Differently named because at this level the language no longer looks remotely like C, the language Stacked splits jumps into two steps: push a label with an `Spushlabel`, and then jump to that label with an `Sjump`. Similarly, it expands internal function calls into pushing the return-label, pushing the arguments, pushing the destination function's label, jumping to that label. At the return-label we have an instruction that stores the return value from the function. This all sounds very complicated until you realize it's exactly how the assembly instructions `jmp` and `call` work.

By this point the compiler has implemented all that Compcert does and dives deeper. This layer implements the function-call mechanism by remembering where to return and jumping to the correct function after pushing the function's arguments and before saving the function's return value.

It also regenerates expressions for the first time since the Clike compilation, because as the name suggests it refers to all local variables (as well as labels for jumping) as being on a stack instead of in a friendly "local environment." Therefore, instead of `Etempvar` taking an identifier as an argument, it simply takes a natural number as a stack offset.

5.7 Ethereum Bytecode

A simple datatype `evm` expresses the different commands that the EVM can actually understand.

These commands include pushing constants, jumping, assigning a value to storage or memory, performing arithmetic on the stack, etc.

This final compilation step performs several operations which should probably be separated for simpler proving:

- Allocate unique identifiers for global variables in Storage.
- Allocate space for hashing and return values in Memory.
- Perform the actual recursive compilation of expressions.
  - Take into account the lack of basic operations in the EVM, for example implementing right-bit-shifts as an exponential followed by a division.
  - Use the types of the arguments to determine whether comparison operations should operate on signed or unsigned words.
- Implement method calls and transfers as `CALL` instructions with different arguments.

6 Semantics

Each language has its own semantics defined in Coq to specify exactly what it means to execute each statement and evaluate each expression. For expressions, the semantics clearly define the value that the expression takes on, but statement semantics are trickier. They are called "smallstep operational semantics," and they can be thought of as the set of steps that a program written in that particular language can take. In Coq they are represented as a relation from start state to end state, where a "State" contains all currently required execution context.

The State includes the values of all local and storage variables, the log of external method calls, the amount of gas remaining, the function and statement currently executing, and the continuation indicating what should come next. For different languages the State may differ: in MiniC the continuation contains both the sequence of successive statements and the call stack of local environments, while in Cgraph a separate list of stackframes contains the previous local environments while the successive statements are implied by the current statement and function. There are often many different kinds of states, like Returnstate which instead of a local environment just remembers a return value, and Callstate which knows a function to call and the arguments to call it with. As an example, here is the definition of states for MiniC:

```
Inductive state: Type :=
  | State
      (f: function)
      (s: statement)
      (k: cont)
      (le: temp_env)
      (se: storage_env)
      (lg: log)
      (gas: nat) : state
  | Callstate
      (fd: fundef)
      (args: list val)
      (k: cont)
      (se: storage_env)
      (lg: log)
      (gas: nat) : state
  | Returnstate
      (res: val)
      (k: cont)
      (se: storage_env)
      (lg: log)
      (gas: nat) : state.
```

Note that the three types of state are used for indicating entry and exit of functions with arguments and return values respectively. The regular state, "State," also contains a function and statement in focus to execute next, as well as the local environment of temporaries with associated values in the scope of the function. Meanwhile, every type of state keeps track of the continuation `k` of where to go next, a storage environment `se`, a log of events `lg`, and an amount of gas still available `gas`.

To give a taste of what each step looks like, here is the first of the step semantics for MiniC, which defines how the `Sassign` statement, which saves the value of an expression into Storage, works.

```
  | step_assign: forall f lv lv_ident rv rv_val k le se lg g,
    eval_lvalue se le lv lv_ident ->
    eval_rvalue se le rv rv_val ->
    step (State f (Sassign lv rv) k le se lg (g + gas_assign true lv rv 2))
      (State f Sskip k le (IdentExtMap.set lv_ident rv_val se) lg g)
```

Note that in a functional language, the plethora of arguments leads to short names, which require some explanation. These lines of code mean that if the left value `lv` evaluates to the pointer `lv_ident` and the right value `rv` evaluates to `rv_val`, within the context of the storage environment `se` and the local environment `le`, then we can execute the step. This step takes a State focused on the statement `(Sassign lv rv)`, with enough gas to actually evaluate the two expressions and jump twice, into a State focused on the no-op statement `Sskip`, now with the storage environment updated correctly and gas reduced. Note in particular that the parts of the state that have not changed: the currently executing function `f`, the list of next things to do `k`, the local environment `le`, and the log of events `lg`.

To give another example from a different part of the compiler, which demonstrates the versatility of these semantics, the following is the semantics for a jump instruction in the Clinear language.

```
  | exec_Sjump: forall s f sd se dst c dstcode lg g,
    find_label dst (fn_code f) = Some dstcode ->
    step (State s f (Sjump dst :: c) sd se lg (g + gas_jump))
      (State s f dstcode sd se lg g)
```

These semantics define what happens in Clinear when looking at the statement `Sjump dst` is followed by any code, provided there is enough gas for a jump and the label `dst` can be found at the start of `dstcode` in the same function. In this case, the program would step to a state where now `dstcode` is in focus and gas is reduced, but everything else remains the same: the list of stackframes `s`, the executing function `f`, the contents of the local environment `sd`, and the log of events `lg`.

The variance in semantics between languages is, to a certain extent, more important than the languages themselves. The descriptions of the languages above are more than just structural, they describe how the languages work, which is exactly the informal definition of semantics for each language.

However, there are still a few critical parts of the semantics which should be described to gain a better understanding of the compiler.

When making external method calls and ether transfers, the semantics rely on a relation defined in the machine environment `me`, which also contains transaction-specific information. Specifically in the semantics of a method call, the precondition looks like the following:

```
(me_callmethod me) a' sg v' args' lg true rvs
```

This precondition states that the method call to address `a'`, for a method with signature `sg`, sending an amount of ether `v'`, taking arguments `args'`, and with a log of previous interactions `lg`, will succeed (`true`) and return values `rvs`. Then the, possibly multiple, return values will get stored in the correct temps, and this call will get appended as an event to the log.

In a different case the semantics appear frustratingly simple because the State doesn't hold as much information as one would like. This is the case of the language Stacked, which doesn't have a list of stackframes in its State because internally it has no concept of a function call. It knows how to push arguments, jump to the correct place, and store return values, but by implementing internal function calls, the language has no way to separately store and reset the local environments of each function. Instead, the proof gets overly complicated, as it requires reasoning about distinct parts of a global stack.

Finally, at the lowest level we had hoped to use the published semantics for the EVM (Hirai). We were able to compile the semantics from the published Lem specification into Coq and include it in our project, but did not get the chance to integrate this with the rest of the project. In general the semantics and proofs were written sequentially, in the downwards direction, so the proofs from MiniC through Clabeled have been completed, and the semantics for Stacked are written, but the lowest level which would include the EVM specification has not yet been breached.

7 Correctness proofs

The purpose of having semantics for each language is to prove their equivalence when we apply the compilation functions. That is, we want the compiled code to act the same way as the original code.

Each compilation step that changes the expressions is accompanied by a proof of expression equivalence. For the MiniC to Clike translation, after proving a few lemmas about pointer arithmetic, the main theorems could be proved. This is the theorem statement for rvalues; the statement is similar for lvalues and for compiling Clike to Stacked expressions.

```
Theorem rvalue_equiv: forall (eMiniC eClike: expr) (result: val),
    clike_rvalue eMiniC = Some eClike ->
    SemanticsMiniC.eval_rvalue me se le eMiniC result ->
    SemanticsClike.eval_rvalue me se le eClike result.
```

This theorem is saying that for any expressions and any result, in the context of a machine environment `me`, a storage environment `se`, and a local environment `le`, if the MiniC expression compiles to the Clike expression via the compilation function `clike_rvalue`, and the MiniC expression evaluates to `result`, then the Clike expression also evaluates to `result`.

Proofs of step semantic equivalence are significantly trickier. This is because the definition of "State" and therefore the definition of "step" depends heavily on the language-specific data types. We would have to define what it means for a State in the abstract syntax tree of Clike to be equivalent to a State in the control flow graph of Cgraph. So that's exactly what we do, with a relation `match_states`. We define such a matching function before proving step equivalence at any stage. For example, here is the first way that MiniC and Clike states can be matched.

```
Inductive match_states: state -> state -> Prop :=
  | match_state: forall f cf s cs k ck le se lg g g'
    (TF: clike_function f = Some cf)
    (TS: clike_stm s = Some cs)
    (TK: match_cont k ck)
    (GAS: g <= g'),
    match_states (State f s k le se lg g) (State cf cs ck le se lg g')
```

As this code illustrates, each part of the state is translated or matched, even between very similar languages. The MiniC function `f` must compile to the Clike function `cf`, and similarly the statements and continuations must match.

When there's no conversion to be done, as is the case with the local environment `se`, the storage environment `se`, and the log `lg`, then we can use the same variable and the states will only match if these components match exactly.

The most interesting feature of this relation is the gas invariance. Instead of asserting that the gas used by each language is always the same, this relation that $g \leq g'$ ensures that the gas remaining in the higher-level language is always weakly *less* than the gas remaining in the lower language. This allows the semantics for higher-level languages to *overestimate* the amount of gas used by each step. The lower-level language like Clinear can specify exactly the amount of gas needed for each jump and conditional jump, but up in MiniC it's easier to say, for example, that evaluating the condition of an `Sifthenelse` statement consumes at most 3 jumps-worth of gas.

Now that the `match_states` relation has been introduced, the step semantic equivalence follows nicely. However, there are a few ways to state the theorem. The Compcert paper proves that all of these ways imply semantic preservation (Leroy).

The compilation from MiniC to Clike is an example of lock-step simulation. A single step is MiniC compiles to exactly one step in Clike. The theorem statement is the following:

```
Theorem step_equiv:
  forall S1 S2, SemanticsMiniC.step me geMiniC S1 S2 ->
  forall T1, match_states S1 T1 ->
  exists T2, SemanticsClike.step me geClike T1 T2 /\ match_states S2 T2.
```

This claims that for any states `S1` and `S2` that MiniC can step between, and any state `T1` of Clike that matches the initial state in MiniC, there exists a final state `T2` in Clike that matches the final state in MiniC, and there's exactly one step in Clike that takes `T1` to `T2`.

That is the simplest type of simulation; a more complicated type is the one from Clike to Cgraph. We want to allow the compiled code to execute multiple steps before reaching the destination state. We also want to allow equivalent states to require no step. To do this we have to define a measure on the states, so that we can't get stuck in equivalent states that loop into each other. The theorem claims the following:

```
Theorem transl_step_correct:
  forall S1 S2, SemanticsClike.step me ge S1 S2 ->
  forall R1, match_states S1 R1 ->
  exists R2,
  (plus (step me) tge R1 R2 \/ (R1 = R2 /\ lt_state S2 S1)) /\ match_states S2 R2.
```

This theorem says that if you can do a step in Clike, you can either stall in Cgraph while the states get smaller, or you can step one or more times (`plus`) to get to the desired state.

The stalling is necessary; for example when Clike steps into a loop, this doesn't correspond to Cgraph doing anything. And allowing multiple steps is also necessary. For example when Clike executes a return statement, that corresponds to two steps in Cgraph. The `Sreturn` pushes the return value and jumps the unique `Sdone` statement that cleans up the function.

The proofs of step equivalence perform a case-analysis on the different types of steps that the source language can take; from there we can prove that the destination language takes an equivalent sequence of steps.

After proving step equivalence, all that remains to prove correctness of the compilation component is to prove that initial states match initial states and final states match final states. Once these simple proofs are handled, the Compcert paper provides a proof that the whole program's semantics are preserved.

8 Results

The proofs of correctness comprise most of the work of this project. The top five of the compiler passes have been proven to preserve semantics. The source language for compilation contains many useful Ethereum constructs, and its semantics include gas requirements. A patched version of DeepSEA can compile to MiniC, but none of the proofs at this top level remain operational.

The most tangible product of my project is a compiler from DeepSEA to Ethereum bytecode. By extracting OCaml code from the Coq verified code, and then composing the compiler with an OCaml pretty-printer I wrote, there is a full pipeline from MiniC to either readable printed bytecode or executable bytes. Multiple test contracts written in MiniC can compile to bytecode. One sample contract sent messages back and forth. Another sample contract computed Fibonacci numbers with a recursive helper function and memoization in persistent storage.

After reading the yellow paper which describes the Ethereum virtual machine (Wood), I wrote an emulator for the EVM. On this emulator, the compiled code successfully ran and computed Fibonacci numbers.

9 Future Work

Several components of this project are unfinished, so finishing them is the first priority in future work.

- The compiler from DeepSEA to MiniC is not proven correct, and DeepSEA does not expose the Ethereum-specific expressions and statements available in MiniC.
- Prove the semantic equivalence of the compilation from Clabeled to Stacked, and from Stacked to bytecode.
  - Possibly introduce new intermediate languages, e.g. for partial-expression evaluation.
  - Incorporate the published Lem formalization of the Ethereum virtual machine
- Write the final proof to glue all of the step simulations together.
- Prove that execution in the final language is deterministic. This requires uniqueness of labels and is required to prove semantic equivalence in both directions.
- Manipulate semantics for calling methods, so as to allow proofs about multiple interacting contracts.

10 References

1. Shao, Zhong. Certik. https://certik.org.
2. Leroy, Xavier. Compcert. https://xavierleroy.org/publi/compcert-backend.pdf
3. Hirai, Yoichi. Ethereum semantics. https://github.com/pirapira/eth-isabelle
4. Wood, Dr. Gavin. Ethereum yellow paper. https://github.com/ethereum/yellowpaper/