Formally Verifying a Payment Channel Smart Contract (Work-in-Progress)

Christopher Fu
Zhong Shao
Vilhelm Sjöberg

December 18, 2017

Abstract

Ethereum is a public, open-source platform that allows users to create contracts that can store data on the Ethereum blockchain and send and receive messages to and from other contracts. Such contracts can be combined in complex ways to create powerful decentralized applications. Because of the nature of the Ethereum blockchain, bugs in contract code can be difficult to spot, hard to rectify, and lead to large monetary losses if exploited. Originally, we proposed to use DeepSEA, a certified programming language developed to allow developers to write specifications for program behavior, to write formally verified smart contracts. In order to do so, we would need to decide on a subset of Solidity to model and an intermediate representation which the DeepSEA compiler frontend would compile into. Over the past semester, I established Coq models for various Ethereum and Solidity constructs, including addresses, integers, mappings, arrays, and signatures. I then used these to model a payment channel system where parties can make off-chain transactions and then resolve disputes via an on-chain contract. Such a payment channel system is useful because it enables parties to quickly perform transactions without waiting for blocks to be mined on the blockchain while also relying on an on-chain contract to ensure that the parties get paid what they are owed. Moving forward, I aim to prove the correctness of my payment channel model and look to reconcile the work I have done with the original stated goal of establishing a useful intermediate representation for the DeepSEA compiler frontend.
1 Introduction to Ethereum

Ethereum is a public, open-source platform that allows users to build and run decentralized applications that run on blockchain technology [10]. A blockchain is a distributed computing architecture where each node in the network executes and records transactions as blocks in the blockchain. Because only one block is added at a time and each block contains a cryptographic proof of its validity, each node in the network can agree upon the current state of the blockchain.

The blockchain forms the backbone of Bitcoin, a cryptocurrency released in 2009, which uses it to record transactions between users [11]. Ethereum, on the other hand, keeps track of the state of every account on the blockchain.

The account is the basic building block of Ethereum, of which there are two types: externally owned accounts (EOAs) and contract accounts (contracts) [1, 3]. EOAs are human-controlled because they are controlled by private keys held by the accounts owner. Contracts (often also referred to as smart contracts), on the other hand, are a collection of code and data that reside at a specific address on the Ethereum blockchain in a binary format called Ethereum Virtual Machine (EVM) bytecode. Contracts, which are typically written in a higher-level language like Solidity and compiled into EVM bytecode to be deployed to the blockchain, can send and receive messages to and from each other and to accounts. In this manner, they can be put together to build decentralized applications backed by the Ethereum blockchain. Examples of such applications range from voting applications, to peer-to-peer trading markets, to video games.

2 Original project proposal

Originally, we proposed to use DeepSEA, a certified programming language developed to allow developers to write specifications for program behavior [13], to write formally verified smart contracts. This way, we could guarantee that the behavior of such contracts will be well-defined and bug-free. DeepSEA was originally used to aid in the development of mCertiKOS, a fully verified hypervisor that can boot a version of Linux as a guest. Edsger, the DeepSEA compiler frontend, and a Coq-CompCertX backend are used to compile DeepSEA specifications into certified executables. Coq is a formal proof assistant that provides a language to write programs and prove properties about them, and CompCertX is a variant of CompCert, a formally verified C compiler that can generate code for x86, PowerPC, and ARM processors.

We aimed to generate Coq specifications from DeepSEA specifications of Solidity contracts. This would have allowed us to prove properties
of DeepSEA contract specifications in Coq. We first looked to model the structure of some typical Solidity contracts to determine what some useful properties to prove about contracts looked like. This way, we hoped to be able to gain a better understanding of what a potential intermediate representation for the DeepSEA frontend would look like.

3 Modeling the ERC20 Token Standard

I first started by looking at the ERC20 Token Standard, which is a standard API for the transfer of tokens [4]. Once I wrote a Coq specification for ERC20, I would then be able to prove some properties about it, such as the fact that the total amount of tokens held by all users is constant over any sequence of transactions.

The main functions of the ERC20 standard are `transfer()` and `approve()`, which allow users to transfer tokens and approve token withdrawals, respectively. Their function signatures look like the following in Solidity:

```solidity
function transfer(address _to, uint256 _value) returns (bool success)
function approve(address _spender, uint256 _value) returns (bool success)
```

An example implementation of the ERC20 standard in Solidity is available at [https://github.com/ConsenSys/Tokens/blob/master/contracts/StandardToken.sol](https://github.com/ConsenSys/Tokens/blob/master/contracts/StandardToken.sol). There, we see that the state of the contract (i.e. the balance of tokens for each user) is maintained in two mappings, which are hash tables that return the zero value of the value type when we look up a key that does not exist in the mapping:

```solidity
mapping (address => uint256) balances;
mapping (address => mapping (address => uint256)) allowed;
```

For my ERC20 specification, I needed to model several Solidity concepts in Coq: addresses (20 bytes long), unsigned integers (256 bits long), mappings, messages, and events. I eventually decided to model addresses and integers using the CompCert Integers module [7], which provides a representation of arbitrary size machine integers and operations over them, and mappings using the CompCert Maps module [8], which provides a representation of maps that return a default value when we look up a nonexistent key.

3.1 Addresses and integers

We define a `Wordsize_160` module that will be passed to the `MakeInt` functor to create an `AddressMod` module.
3.2 Mappings

We can now representing mappings keyed on addresses because the Maps module provides an implementation of maps over types that index into positive, which our AddressMod module does.

Module AddressIndexed <: INDEXED_TYPE <: ZeroableType.
Definition t := AddressMod.int.
Definition index (i : AddressMod.int): positive :=
  let (intval, _) := i in
  match intval with
  | Z0 => xH
  | Zpos p => xO p
  | Zneg p => xI p
  end.
Lemma index_inj: forall (x y: AddressMod.int),
  index x = index y -> x = y.
Proof. (* ... *) Qed.
Definition eq := AddressMod.eq_dec.
Definition zero_value := AddressMod.zero.
End AddressIndexed.

Module AddrMap := IMap(AddressIndexed).

(* Now we can make address => uint256 mappings with a 
default value of 0 like this *)
Definition balances := AddrMap.init (Int256Mod.repr 0).

### 3.3 Messages

_Messages_ in Ethereum are passed between accounts and are used to transfer data and ether. In Solidity, a message is represented as the variable _msg_ in the global namespace and contains members such as _sender_, _data_, _gas_, and _value_ [9]. In Coq, we can model this as a record type that is passed to every function. For my current needs, I only need the _sender_ and _value_ fields.

Record Msg := { sender : Address; value : Z }.
Record State := { (* contract state *) }.

Definition transfer_from_owner (st : State) (msg : Msg)
  (to : Address) (amount : Int256) : option State :=
  (* ... *)

### 3.4 Events

_Events_ in Ethereum are emitted by contracts, and clients connected to the Ethereum JSON-RPC API, typically JavaScript distributed application (dapp) frontends, can listen to events and act accordingly. For now, I record events in a _GlobalState_ record, which gets passed around and modified as needed by contract functions.

Inductive Event :=
  | EventUpdate : Z -> Event
  | EventPending : Z -> Z -> Event
  | EventInit
  | (* ... *)

Record GlobalState := { events : list Event;
  (* other state *) }.

5
4 Pivoting to payment channels

At this point, we decided to shift our efforts to modeling payment channels because we wanted to examine a more complex and more practically useful system of smart contracts. A payment channel is an off-chain mechanism for performing transfers of credit between two parties (or players) [12]. The process of performing a transaction over a payment channel looks roughly like the following:

(a) **Open the channel.** Both parties deposit some amount of credit with an on-chain transaction as a security deposit.

(b) **Make off-chain transactions.** The parties can now exchange signed messages to pay each other. Each party keeps track of the current balance.

(c) **Finalize.** Once a party decides to withdraw its balance, it notifies the on-chain contract that the payment channel will be closed and submits the proposed payouts. The contract then notifies the other party and waits for a fixed amount of time for any disputes. If there is a dispute between the parties about the amounts paid, both parties can send evidence (e.g. the signed messages) to the on-chain contract, which then resolves the dispute based on the evidence.

The main benefit of using payment channels instead of directly performing transactions on the blockchain is that the parties do not have to wait for the block containing the transaction to be mined for the payment to be “confirmed.” Currently, the amount of time it takes for an Ethereum block to be mined is around 12 seconds [6], which is too slow to facilitate high-frequency transactions. Using a payment channel, two parties can exchange many off-chain transactions, thereby sidestepping the block time issue, and then take advantage of the blockchain to pay out credits and resolve disputes at the very end.

I started by looking at the code provided by the Miller 2017 paper at [https://github.com/amiller/sprites](https://github.com/amiller/sprites) I focused on `contractPay.sol` and `test_pay.py`, which are the implementation of the on-chain contract and some test driver code, respectively. In the process, I had to model new data structures and concepts.

4.1 Arrays

In Solidity, arrays can either be fixed-length or dynamic [2]. `contractPay.sol` only uses fixed-length arrays, so for now I only model those. The `Array` module models arrays as a tuple of (length × `ZTree`) (`ZTree` is provided by the CompCert Maps module). Elements at index `i` are located at key `i` in the `ZTree`. When we initialize an array of length `l`, we create a
Module Array(T : ZeroableType).
    Definition t : Type := (Z * ZTree.t T.t).
    Definition get (a : t) (i : Z) : option T.t := (*
                             ... *)
    Definition set (a : t) (i : Z) (v : T.t) : option t
                             := (* ... *)
    Definition init (elems : list T.t) : option t := (*
                             ... *)
    (* ... *)
End Array.

4.2 Hashes and signatures

The on-chain contract depends on receiving signed messages to verify proposed payouts when a payment channel is finalized. In Solidity, the `ecrecover()` function recovers the address associated with the public key from an elliptic curve signature [5]. In Coq, I model a signature for now as a simple inductive type with a single constructor, and a corresponding `verify_signature()` function that destructs the signature to determine who signed it.

Inductive Hash :=
| Hash3 : Z -> IntArr.t -> IntArr.t -> Hash
| Hash2Z : Z -> Z -> Hash
| Hash3Z : Z -> Z -> Z -> Hash
| (* ... *)

Inductive Sig :=
| Signature : Hash -> Address -> Sig.
Definition sign (h : Hash) (addr : Address) :=
    Signature h addr.
Definition verify_signature (addr : Address) (hash :
    Hash) (sig : Sig) : bool :=
    match sig with
    | Signature h a => AddressMod.eq addr a
end.

For our correctness proofs later, we will also need a way to ensure that one party cannot impersonate another party and sign a message using another address.
4.3 Proof of correctness of player and on-chain contract models (work-in-progress)

Once I wrote models for the player and on-chain contract code, we could proceed to prove the correctness of the models. We adopted an asymmetric proof strategy, where we have an honest player (L for left) communicating with a possibly dishonest player (R for right).

If we assume that the only actions that players can perform are pay and trigger (i.e. finalize), if the player model keeps track of all signed transactions so far (received and sent), L can always at least receive what he is owed. If R triggers and proposes a payout, and L agrees, no action needs to be taken by L. If R proposes a payout that L doesn’t agree with, which may occur if R omits one or more payment transactions, then L can provide the contract with the signed payments. And since R can’t sign a payment transaction to make it look like it came from L, L will always get at least what he is owed. R could add extra payments from R to L, but this would result in L receive more than he is owed, which is fine.

More concretely, we can represent the possible steps that players may take as an inductive relation: L could pay R, or R could pay L.

Inductive Side := Left | Right.

Inductive step : Side ->
   (PlayerState * ContractState * (list (Transaction * Sig))) ->
   (PlayerState * ContractState * (list (Transaction * Sig))) ->

Prop :=
| left_pay : forall pst cst msgs (pst’ : PlayerState)
  (cst’ : ContractState) amt msg pst’ deposit_l
      deposit_r,
      IntArr.get (PmtChan.deposits cst) 0 = Some
      deposit_l ->
      IntArr.get (PmtChan.deposits cst) 1 = Some
      deposit_r ->
      Player.pay pst cst amt = Some (msg, pst’) ->
      step Left (pst, cst, msgs) (pst’, cst, msgs ++ [msg])

| right_pay : forall pst cst msgs (pst’ : PlayerState)
  (cst’ : ContractState) rd amt msg pst’ tr
      deposit_l deposit_r,
      IntArr.get (PmtChan.deposits cst) 0 = Some
      deposit_l ->
      IntArr.get (PmtChan.deposits cst) 1 = Some
      deposit_r ->
      tr = Pay 1 rd amt ->
Then, for any step that $L$ takes, for any subsequent step that $R$ takes, $L$ should be able to withdraw at least the amount that he is owed. We can express this property in Coq as follows:

\[
\text{Inductive can_force (amt : \mathbb{Z}) (st : PlayerState * ContractState * (list (Transaction * Sig)))): Prop :=}
\]

\[
| \text{left_step : forall s s', step Left s s' ->}
\text{forall s'', step Right s' s'' -> can_force amt s}'' ->
\text{can_force amt st}.
\]

\[
| \text{finished : forall deposit credit,}
\text{Some deposit = IntArr.get (PmtChan.deposits (snd (fst st))) 0 ->}
\text{Some credit = IntArr.get (PmtChan.credits (snd (fst st))) 0 ->}
\text{deposit + credit >= amt -> can_force amt st.}
\]

Currently, I am working on proving this property.

## 5 Conclusion

So far, I have established some useful foundations for modeling Solidity contracts in Coq. Various basic pieces like mappings, arrays, and signatures have Coq counterparts that we can use to model useful contracts, such as payment channels. Moving forward, I will aim to complete the proof of correctness of my payment channel model and look to reconcile the work I have done with the original stated goal of establishing a useful intermediate representation for the DeepSEA compiler frontend.

### References


