Smart Contract Interactions in Coq

Jakob Botsch Nielsen and Bas Spitters

Concordium Blockchain Research Center, Computer Science, Aarhus University

Abstract. We present a model/executable specification of smart contract execution in Coq. Our formalization allows for inter-contract communication and generalizes existing work by allowing modelling of both depth-first execution blockchains (like Ethereum) and breadth-first execution blockchains (like Tezos). We represent smart contracts programs in Coq's functional language Gallina, enabling easier reasoning about functional correctness of concrete contracts than other approaches. In particular we develop a Congress contract in this style. This contract – a simplified version of the infamous DAO – is interesting because of its very dynamic communication pattern with other contracts. We give a high-level partial specification of the Congress's behavior, related to reentrancy, and prove that the Congress satisfies it for all possible smart contract execution orders.

1 Introduction

Since Ethereum, blockchains make a clear separation between the consensus layer and the execution of smart contracts. In Ethereum's Solidity language contracts can arbitrarily call into other contracts as regular function calls. Modern blockchains further separate the top layer in an execution layer and a contract layer. The execution layer schedules the calls between the contracts and the contract layer executes individual programs. The choice of execution order differs between blockchains. For example, in Ethereum the execution is done in a synchronous (or depth first) order: a call completes fully before the parent continues, and the parent is able to observe its result. Tezos uses the breadth first order.

We provide[1] a model/executable specification of the execution and contract layer of a third generation blockchain in the Coq proof assistant. We use Coq's embedded functional language Gallina to model contracts and the execution layer. This language can be extracted to certified programs in for example Haskell or Ocaml. Coq's expressive logic also allows us to write concise proofs. The consensus protocol provides a consistent global state which we treat abstractly in our formalization.

We work with an account-based model. We could also model the UTxO model by converting a list of UTxO transactions to a list of account transactions [Zah18]. Like that work, we do not model the cryptographic aspects, only the accounting aspects: the transactions and contract calls.

[1] `https://gitlab.au.dk/concordium/smart-contract-interactions/tree/v1.0`

This text is organized as follows: Section 2 describes the implementation of the execution layer in Coq. In Section 3 we provide a simple principled specification for the Congress. By using such specifications one avoids having to deal with reentrancy bugs in a post-hoc way. Section 4 discusses related work. Section 5 concludes.

2 Implementation

2.1 Basic assumptions

Our goal is to model a realistic blockchain with smart contracts. To do so we will require this blockchain to supply some basic operations that are to be used both by smart contracts and when specifying our semantics. Our most basic assumptions are captured as a typeclass:

```
Class ChainBase :=
  { Address : Type;
    address_countable :> Countable Address;
    address_is_contract : Address → bool;
    ...  }.
```

Specifically we require a countable `Address` type with a clear separation between addresses belonging to contracts and to users. While this separation is not provided in Ethereum its omission has led to exploits before[2] and we thus view it as realistic that future blockchains allow this. Other blockchains commonly provide this by using some specific format for contract addresses, for example, Bitcoin marks addresses with associated scripts using so-called pay-to-script-hash addresses which always start with 3.

Generally all semantics and smart contracts will be abstracted over an instance of this type, so in the following sections we will assume we are given such an instance.

2.2 Smart Contracts

We will consider a functional smart contract language. Instead of modelling the language as an abstract syntax tree in Coq, as in [AS19], we model individual smart contracts as records with (Coq) functions.

Local state. It is not immediately clear how to represent smart contracts by functions. For one, smart contracts have local state that they should be able to access and update during execution. In Solidity, the language typically used in Ethereum, this state is mutable and can be changed at any point in time. It is possible to accomplish something similar in pure languages, for example by using a state monad which allows state to be updated at any point during a

[2] See for instance `https://www.reddit.com/r/ethereum/comments/916xni/how_to_pwn_fomo3d_a_beginners_guide/`

function's execution, but we do not take this approach. Instead we use a more traditional functional approach where the contract takes as input its current state, and returns a single new, updated state.

However, different contracts will typically have different types of states. A crowdfunding contract may wish to store a map of backers in its state while an auction contract would store information about ongoing auctions. To facilitate this polymorphism we use an intermediate storage type called `SerializedValue`. We define conversions between `SerializedValue` and primitive types like booleans and integers plus derived types like pairs, sums and lists. Generally this allows conversion from and to `SerializedValue` to be handled implicitly and mostly transparently to the user.

Inter-contract communication. In addition to local state we also need some way to handle inter-contract communication. In Solidity contracts can arbitrarily call into other contracts as regular function calls. This would once again be possible with a monadic style, for example by the use of a promise monad where the contract would ask to be resumed after a call to another contract had finished. To ease reasoning we choose a simpler approach where contracts return actions that indicate how they would like to interact with the blockchain, allowing transfers, contract calls and contract deployments only at the end of execution. The blockchain will then be responsible for scheduling these actions in its execution layer.

With this design we get a clear separation between contracts and their interaction with the chain. That such separations are important has been realized before, for instance in the design of Michelson and Scilla [SKH18a]. Indeed, a ”tail-call” approach like this forces the programmer to update the contract's internal state before making calls to other contracts, mitigating by construction reentrancy issues such as the infamous DAO exploit.

Thus, contracts will take their local state and some data allowing them to query the blockchain. As a result they then optionally return the new state and some operations (such as calls to other contract) allowing inter-contract communication. Overall, this design is very similar to the Tezos blockchain where contracts are written in Michelson which follows a similar approach.

The Ethereum model may be compared to object-oriented programming. Our model is similar to the actor model, as contracts do not read or write the state of another contract directly, but instead communicate via messages. One finds similar models in Liquidity and in Scilla, which is based on IO-automata.

The contract's view. Smart contracts are typically allowed to query various data about the blockchain during execution, such as the current block height. Normally this is provided as special instructions. For instance, this is the case in EVM bytecode used for Ethereum. Since we use a shallow embedding we will instead pass this as an additional argument to the contract. In our framework, we give contracts the following view of the blockchain:

```
Definition Amount := Z.
```

```
Record Chain :=
  { chain_height : nat;
    current_slot : nat;
    finalized_height : nat;
    account_balance : Address → Amount; }.
```

We allow contracts to access basic details about the blockchain, like the current chain height, slot number and the finalized height. The slot number is meant to be used to track the progression of time; in each slot, a block can be created, but it does not have to be. The finalized height allows contracts to track the current status of the finalization layer available in for example the Concordium blockchain [MMNT19]. This height is different from the chain height in that it guarantees that blocks before it will not be changed. We finally also allow the contract to access balances of accounts, as is common from other blockchains.

The contract. The final piece of information provided to contracts when they are executed is information about the call. Overall, we thus represent contracts using the following types:

```
Record ContractCallContext :=
  { ctx_from : Address;
    ctx_contract_address : Address;
    ctx_amount : Amount; }.
Inductive ActionBody :=
  | act_transfer (to : Address) (amount : Amount)
  | act_call (to : Address) (amount : Amount)
             (msg : SerializedValue)
  | act_deploy (amount : Amount) (c : WeakContract)
               (setup : SerializedValue)
with WeakContract :=
  | build_weak_contract
      (init : Chain → ContractCallContext →
         SerializedValue (* setup *) →
         option SerializedValue)
      (receive : Chain → ContractCallContext →
         SerializedValue (* state *) →
         option SerializedValue (* message *) →
         option (SerializedValue * list ActionBody)).
```

Here the `ContractCallContext` type represents information that is common to when the contract executed due to deployment or due to receiving a message. It contains the source address (`ctx_from`), the contract's own address (`ctx_contract_address`) and the amount of money transferred (`ctx_amount`). The `ActionBody` type represents operations that interact with the chain. It allows for simple messageless transfers (`act_transfer`), calls with messages (`act_call`), and deployment of new contracts (`act_deploy`). These do not contain a source address to model that while contracts can interact with the blockchain, they do not get to specify the source (which is their own address) when they do so. Finally, a contract is two functions. The `init` function is used when a contract is deployed to set up its initial state, while the `receive` function will be used for transfers and calls with messages afterwards. They both return option types, allowing the contract to signal invalid calls or deployments. The `receive` function additionally returns a list of `ActionBody` that it wants to be performed in the chain after, as we described above. Later, we will also use a representation where there *is* a source address; we call this type `Action`:

```
Record Action :=
  { act_from : Address;
    act_body : ActionBody; }.
```

This type might resemble what is normally called a transaction, but we make a distinction between the two. An `Action` is an unevaluated operation that, when executed by an implementation, affects the blockchain's state. Particularly, compared to a transaction it is underrepresented in that `act_deploy` does not contain the address of the contract to be deployed. This models that it is the implementation that picks the address of a newly deployed contract, not the contract making the deployment. We will later describe our `ActionEvaluation` type which captures more in depth the choices made by the implementation while executing an action.

The functions of contracts may seem peculiar in that they are typed using `SerializedValue` parameters. This is also the reason for the name `WeakContract`. Generally this makes specifying semantics simpler, since the semantics can deal with contracts in a generic way. However, for users of the framework writing concrete contracts this form of ”string-typing” makes things harder. For this reason we provide a dual notion of a *strong* contract, which is a polymorphic version of contracts generalized over the setup, state and message types. Users of the framework will only need to be aware of this notion of contract, which does not contain references to `SerializedValue` at all.

One could also imagine an alternative representation using a dependent record of setup, state and message types plus functions over those types. However, such a representation makes it nearly impossible for contracts to interact with other contracts since they will somehow need to prove that the messages they are sending are of the types stored in this record. In particular this is difficult when the blockchain has no knowledge about individual contracts and only works generically with them.

2.3 Semantics

Next we wish to specify the semantics of the execution layer.

Environments. The `Chain` type given above is merely the contract's view of the blockchain and does not store enough information to allow the blockchain to run actions. More specifically we need to be able to look up information about currently deployed contracts like their functions and state. We augment the `Chain` type with this information and call it an `Environment`:

```
Record Environment :=
  { env_chain :> Chain;
    env_contracts : Address → option WeakContract;
    env_contract_states :
      Address → option SerializedValue; }.
```

It is not hard to define functions that allow us to make updates to environments. For instance, inserting a new contract is done by creating a new function that checks if the address matches and otherwise uses the old map. In other words we use simple linear maps in the semantics. In similar ways we can update the rest of the fields of the `Environment` record.

Evaluation of actions. When contracts return actions the execution layer will somehow need to evaluate the effects of these actions. We define this as a ”proof-relevant” relation `ActionEvaluation` in Coq:

```
ActionEvaluation : Environment → Action →
  Environment → list Action → Type
```

This relation captures the requirements and effects of executing the action in the environment. It is ”proof-relevant”, meaning that it can be inspected, which is useful since actions by themselves are underspecified. For example, a contract can return an action that deploys a new contract; in this case we leave it up to the implementation to pick an appropriate address for the new contract. However, when reasoning about action evaluation it is useful to know which address a contract was deployed to and this information can be retrieved by inspecting the evaluation.

We define the relation by three cases: one for transfers of money, one for deployment of new contracts, and one for calls to existing contracts. To exemplify this relation we give its formal details for the simple transfer case below:

```
| eval_transfer :
    forall {pre : Environment}
           {act : Action}
           {new_env : Environment}
           (from to : Address)
           (amount : Amount),
      amount ≤ account_balance pre from →
      address_is_contract to = false →
      act_from act = from →
      act_body act = act_transfer to amount →
      EnvironmentEquiv
        new_env
        (transfer_balance from to amount pre) →
      ActionEvaluation pre act new_env []
```

In this case the sender must have enough money and the recipient cannot be a contract. When this is the case a transfer action and the old environment evaluate to the new environment where the `account_balance` has been updated appropriately. Finally, such a transfer does not result in more actions to execute since it is not associated with execution of contracts. Note that we close the evaluation relation under extensional equality (`EnvironmentEquiv`).

We denote this relation by the notation $\langle \sigma, a \rangle \Downarrow (\sigma', l)$. The intuitive understanding of this notation is that evaluating the action a in environment σ results in a new environment σ' and new actions to execute l.

Chain traces. The `Environment` type captures enough information to evaluate actions. We further augment this type to keep track of the queue of actions to execute. In languages like Solidity this data is encoded implicitly in the call stack, but since interactions with the blockchain are explicit in our framework we keep track of it explicitly in the `ChainState` type.

```
Record ChainState :=
  { chain_state_env :> Environment;
    chain_state_queue : list Action; }.
```

We are now ready to define what it means for the chain to take a step. Formally, this is defined as a ”proof-relevant” relation `ChainStep`:

```
ChainStep : ChainState → ChainState → Type
```

We denote this relation with the notation $(\sigma, l) \to (\sigma', l')$, meaning that we can step from the environment σ and list of actions l to the environment σ' and list of actions l'. We give this relation as simplified inference rules below.

$$\textsc{step-block}\quad \frac{b \text{ valid in } \sigma \qquad \mathit{acts} \text{ from users}}{(\sigma, \texttt{[]}) \to (\texttt{add\_block}\ b\ \sigma, \mathit{acts})}$$

$$\textsc{step-action}\quad \frac{\langle \sigma, a \rangle \Downarrow (\sigma', l)}{(\sigma, a :: l') \to (\sigma', l \mathbin{++} l')}$$

$$\textsc{step-permute}\quad \frac{\mathrm{Perm}(l, l')}{(\sigma, l) \to (\sigma, l')}$$

The <span style="font-variant:small-caps">step-block</span> rule allows the addition of a new block with associated actions. This is the only way to add new actions into a trace when the queue is empty. We require that the block information (b in the rule) is valid in the current environment (the b valid in σ premise), meaning that it needs to satisfy some well-formedness conditions. For example, if the chain currently has height n, the next block added needs to have height $n + 1$. There are other well-formedness conditions on other fields, such as the finalized height, but we omit them here for brevity. Another condition is that all added actions must come from users (the *acts* from users premise). This models the real world where transactions added in blocks are ”root transactions” from users, and carrying out these transactions might cause contracts to generate new transactions. In our model this condition is crucial to ensure that transfers from contracts can happen only due to execution of their associated code. When the premises are met we update information about the current block (such as the current height and the balance of the creator, signified by the `add_block` function) and update that the queue now contains the actions that were added.

The <span style="font-variant:small-caps">step-action</span> rule allows the evaluation of the action in the beginning of the queue, replacing it with the resulting new actions to execute. This new list (l in the rule) is concatenated at the beginning, corresponding to using the queue as a stack. This results in a depth-first execution order of actions. The STEP-PERMUTE rule allows an implementation to use a different order of reduction by permuting the queue at any time. For example, it is possible to obtain a breadth-first order of execution by permuting the queue so that newly added events are in the back. In this case the queue will be used like an actual FIFO queue.

Building upon steps we can further define *traces* as the proof-relevant reflexive transitive closure of the step relation. In other words, this is a sequence of steps where each step starts in the state that the previous step ended in. Intuitively the existence of a trace between two states means that there is a semantically correct way to go between those states. If we let ε denote the empty environment with no queue this allows us to define a concept of *reachability*. Formally we say a state (σ, l) is *reachable* if there exists a trace starting in ε and ending in (σ, l). In Coq we define this as

```
Definition reachable (state : ChainState) : Prop :=
  inhabited (ChainTrace empty_state state).
```

Generally, only reachable states are interesting to consider and most proofs are by induction over the trace to a reachable state.

2.4 Building blockchains

We connect our semantics to an executable definition of a blockchain with a typeclass in Coq:

```
Class ChainBuilderType := {
    builder_type : Type;
    builder_initial : builder_type;
    builder_env : builder_type → Environment;
    builder_add_block
      (builder : builder_type)
      (header : BlockHeader)
      (actions : list Action) :
      option builder_type;
    builder_trace (builder : builder_type) :
      ChainTrace empty_state
      (build_chain_state (builder_env builder) []);}.
```

A chain builder is a dependent record consisting of an implementation type (`builder_type`) and several fields using this type. It must provide an initial builder, which typically would be an empty chain, or a chain containing just a genesis block. It must also be convertible to an environment allowing to query various information about the state. Furthermore, it must define a function that allows addition of new blocks. Finally, the implementation needs to be able to give a trace showing that the current environment is reachable with no more actions left in the queue to execute. This trace captures a definition of soundness, since it means that the state of such a chain builder will always be reachable[3].

Instantiations. A priori it is not a guarantee that the semantics we have defined are reasonable. More formally it is possible that `ChainBuilderType` is uninhabited which makes proving properties based on it uninteresting. Thus, as a sanity check, we implement two instances of this typeclass. Both of our implementations are based on finite maps from the std++ library used by Iris [JKJ+18] and are thus relatively efficient compared to the linear maps used to specify the semantics. The difference in the implementations lies in their execution model: one implementation uses a depth-first execution order, while the other uses a breadth-first execution order. The former execution model is similar to the EVM while the latter is similar to Tezos.

These implementations are useful as sanity checks but they also serve other useful purposes in the framework. Since they are executable they can be used to test concrete contracts that have been written in Coq. This involves writing the contracts and executing them using Coq's `Compute` vernacular. In addition, they can also be used to give counter-examples to properties. In the next section we will introduce the *Congress* contract, and we have used the depth-first implementation of our semantics to formally show that this contract with a small change is vulnerable to reentrancy.

3 Case: Congress – a simplified DAO

In this section we will present a case study of implementing and partially specifying a complex contract in our framework.

3.1 The Congress contract

Wang [Wan18] gives a list of eight interesting Ethereum contracts. One of these is the so-called *Congress* in which members of the contract vote on *proposals*. Proposals contain transactions that, if the proposal succeeds, are sent out by the Congress. These transactions are typically monetary amounts sent out to some address, but they can also be arbitrary calls to any other contract.

We pick the Congress contract because of its complex dynamic interaction pattern with the blockchain and because of its similarity to the infamous DAO contract that was deployed on the Ethereum blockchain and which was eventually hacked by a clever attacker exploiting reentrancy in the EVM.

The Congress can be seen as the core of the DAO contract, with the DAO implementing various additional mechanisms on top of voting for proposals. For example, proposals can be seen as investments into other projects, and the DAO contract kept track of the voters on each proposal to be able to pay back rewards to these people in case the project turned out successful.

[3] We do not currently include a notion of completeness. For instance, it is possible to define a trivial chain builder that just ignores the blocks and actions to be added.

We implement the logic of the Congress in roughly 150 lines of Gallina code. The type of messages accepted by the Congress can be thought of as its interface since this is how actors on the blockchain can interact with it. For the Congress we define the following messages:

```
Inductive Msg :=
  | transfer_ownership : Address → Msg
  | change_rules : Rules → Msg
  | add_member : Address → Msg
  | remove_member : Address → Msg
  | create_proposal : list CongressAction → Msg
  | vote_for_proposal : ProposalId → Msg
  | vote_against_proposal : ProposalId → Msg
  | retract_vote : ProposalId → Msg
  | finish_proposal : ProposalId → Msg.
```

The Congress has an owner who is responsible for managing the rules of the congress and the member list. By default, we set this to be the creator of the congress. The owner can transfer his ownership away with the `transfer_ownership` message. For example, it is possible to make the Congress its own owner, in which case all rule changes and modifications to the member list must happen through proposals (essentially making the Congress a democracy).

Anyone can use the `create_proposal` and `finish_proposal` messages. We allow proposals to contain any number of actions to send out, though we restrict the actions to only transfers and contract calls (i.e. no contract deployments). This restriction is necessary because this would require the state of the Congress to contain the contracts to deploy. Since contracts are functions in our shallow embedding this would require storing higher order state which we do not allow in the framework. This is a downside to the shallow embedding – with a deep embedding like [AS19], the code could be stored as an AST or bytes.

While proposals can be finished by anyone they must first have been debated for some period specified in the rules of the congress. During this period, members of the congress have the ability to vote for or against the proposal. After the debating period is over the proposal can be finished and the Congress will remove it from its internal storage and send out its actions in case it passed. The conditions for passing are once again specified in the rules, which contain values such as the margin of yes-votes required.

3.2 A partial specification

The vulnerability of the DAO was in reward payout code in which a specially crafted contract could reenter the DAO causing it to perform actions an unintended number of times. Specifically, the attacker was able to propose a so-called *split* and have the original DAO transfer a disproportionate amount of money to a new DAO contract under his control. The Congress does not contain similar code, but the same kind of bug would be possible in code responsible for carrying out proposals.

Previous research has focused on defining this kind of reentrancy formally which we could also define and prove in our framework. Such (hyper-)properties are interesting, but they also rely heavily on the benefit of hindsight and their statements are complex and hard to understand. Instead we would like to come up with a natural specification pertaining to the Congress that a programmer could reasonably have come up with, even without knowledge of reentrancy. Our goal with this is to apply the framework in a very concrete setting.

The specification we give is based on the following observation: any transaction sent out by the congress should correspond to an action that was previously created with a `create_proposal` message. This is a temporal property because it says something about the past whenever an outgoing transaction is observed. Temporal logic is not natively supported by Coq, so this would require some work. Therefore we prefer a similar but simpler property: the number of actions in previous `create_proposal` messages is always greater than or equal to the total number of transactions the congress has sent out. This is not a full specification of the behavior of the Congress but proving this property can help increase trust that the congress is not vulnerable to reentrancy. With such a proof, any bug exploiting the Congress in a similar way to the DAO would somehow require a new proposal to be created for each time the exploit was carried out. In particular, the result would not have been provable in the original DAO contract because of the reentrancy exploit. Our main result about the congress is a formal proof that this always holds after adding a block:

```
Corollary congress_txs_after_block
          {ChainBuilder : ChainBuilderType}
          prev creator header acts new :
  builder_add_block prev creator header acts = Some new →
  forall addr,
    env_contracts new addr =
    Some (Congress.contract : WeakContract) →
    length (outgoing_txs (builder_trace new) addr) ≤
    num_acts_created_in_proposals
      (incoming_txs (builder_trace new) addr).
```

This result states that, after adding a block, any address at which a Congress contract is deployed satisfies the property previously described. Here the function `num_acts_created_in_proposals` looks at all previous `create_proposal` messages and sums the number of actions in them. The `incoming_txs` and `outgoing_txs` functions are general functions that finds transactions (evaluation of actions) in a trace. In this sense the property treats the contract as a black box, stating only things about the transactions that has been observed on the blockchain.

We prove this property by generalizing it and proving something stronger. Specifically, instead of stating the invariant over just the transactions and proposals we state it over the following data:

- The internal state of the contract; more specifically, the current number of actions in proposals stored in the internal state.
- The number of transactions sent out by the Congress, as before.

- The number of actions *in the queue* where the Congress is the source.
- The number of actions created in proposals, as before.

The key observations being that

1. When a proposal is created, the number of actions created in proposals goes up, but so does the number of actions in the internal state of the Congress.
2. When a proposal is finished, the number of actions in the internal state goes down, but the number of actions in the queue goes up accordingly (assuming the proposal was voted for). In other words, actions "move" from the Congress's internal state to the queue.
3. When an outgoing transaction appears on the chain it is because an action moved out of the queue.

Especially observation 3 is interesting. It allows us to connect the evaluation of a contract in the past to its resulting transactions on the chain, even though these steps can be separated by many unrelated steps in the trace.

The proof of the stronger statement is straightforward by inducting over the trace and showing that it always holds. When deploying the Congress we need to establish the invariant which boils down to proving functional correctness of the `init` function and the usage of some results that hold for contracts which have just been deployed (for instance, such contracts have not made any outgoing transactions). On calls to the Congress the invariant needs to be reestablished, which boils down to proving functional correctness of the `receive` function. Crucially, we can reestablish the invariant because the implementation of the Congress clears out proposals from its state *before* the actions in the proposal are evaluated (the DAO was vulnerable because it neglected to do this on splits). Once we have established this stronger statement the result easily follows as a direct corollary.

4 Related work

Both Simplicity [O'C17] and Scilla [SKH18a] are smart contract languages with an embedding in Coq. Temporal properties of several smart contracts have been verified in Scilla [SKH18b], although our congress contract is more complex than the contracts described in that paper. We are unaware of an implementation of such a contract in Scilla. Scilla, as an intermediate language which includes both a functional part and contract calls, uses a CPS translation to ensure that every call to another contract is done as the last instruction. In our model, the high-level language and the execution layer are strictly separated.

The formalization of the EVM in F* [GMS18] can be extracted and used to run EVM tests to show that it is a faithful model of the EVM. However, they do not prove properties of any concrete contracts. Instead they consider classes of bugs in smart contracts and try to define general properties that prevent these. One of these properties, call integrity, is motivated by the DAO and attempts to capture reentrancy. Intuitively a contract satisfies call integrity if the calls it makes cannot be affected by code of other contracts. VerX [PDT$^+$19] uses temporal logic and model checking to check a similar property. Such statements are not hard to state in our framework given Coq's expressive logic, and it seems this would be an appropriate property to verify for the Congress. Unfortunately even a correct Congress does not satisfy this property, since it is possible for called contracts to finish proposals which can cause the Congress to perform calls. This property could potentially be proven in a version of the Congress that only allowed proposals to be finished by humans, and not by contracts.

5 Conclusion and future work

We have formalized the execution model of blockchains in Coq and used our formalization to prove formally a result about a concrete contract. Our formalization of blockchain semantics is flexible in that it accounts both for depth-first and breadth-first execution order, generalizing existing blockchains and previous work, while remaining expressive enough to allow us to prove results about complex contracts. We showed for a Congress – a simplified version of the DAO, which still has a complex dynamic interaction pattern – that it will never send out more transactions than have been created in proposals. This is a natural property that aids in increasing trust that this contract is not vulnerable to reentrancy like the DAO.

More smart contracts are available in Wang's PhD thesis [Wan18] and specifying these to gain experience with using the framework will help uncover how the framework itself should be improved. In this area it is also interesting to consider more automatic methods to make proving more productive. For example, temporal logics like LTL or CTL can be useful to specify properties on traces and model checking these can be automated; see e.g. [PDT+19].

Finally, while our current framework is inspired by and generalizes existing blockchains, there is still more work to be done to get closer to practical implementations. Gas is notoriously difficult to deal with in our shallow embedding because tracking costs of operations can not be done automatically, but monadic approaches have been used for similar purposes before [MFN+18]. To deal with this problem we plan to connect our shallow embedding with a deep embedding of the language Oak as described in [AS19], which will also allow proving properties about Oak contracts in our framework. In the other direction it is interesting to consider extraction of our contracts into other languages like Liquidity, Oak or Solidity. This is more directly applicable to current practice.

Acknowledgements We would like to thank the Oak team for stimulating discussions.

References

[AS19] Danil Annenkov and Bas Spitters. Deep and shallow embeddings in Coq. *TYPES*, 2019.

[GMS18] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. A semantic framework for the security analysis of ethereum smart contracts. In *PoST*, pages 243–269. Springer, 2018.

[JKJ+18] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28, 2018.

[MFN+18] Jay McCarthy, Burke Fetscher, Max S New, Daniel Feltey, and Robert Bruce Findler. A Coq library for internal verification of running-times. *Science of Computer Programming*, 164:49–65, 2018.

[MMNT19] Bernardo Magri, Christian Matt, Jesper Buus Nielsen, and Daniel Tschudi. Afgjort – a semi-synchronous finality layer for blockchains. Cryptology ePrint 2019/504, 2019. `https://eprint.iacr.org/2019/504`.

[O'C17] Russell O'Connor. Simplicity: A new language for blockchains. *CoRR/1711.03028*, 2017.

[PDT+19] Anton Permenev, Dimitar Dimitrov, Petar Tsankov, Dana Drachsler-Cohen, and Martin Vechev. Verx: Safety verification of smart contracts. *Security and Privacy 2020*, 2019.

[SKH18a] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *arXiv:1801.00687*, 2018.

[SKH18b] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Temporal properties of smart contracts. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice*, pages 323–338. Springer, 2018.

[Wan18] Peng Wang. *Type System for Resource Bounds with Type-Preserving Compilation*. PhD thesis, MIT, 2018.

[Zah18] Joachim Zahnentferner. Chimeric ledgers: Translating and unifying utxo-based and account-based cryptocurrencies. Cryptology ePrint 2018/262, 2018. `https://eprint.iacr.org/2018/262`.