Abstract

This paper outlines key design principles of Scilla—an intermediate-level language for verified smart contracts.

Scilla provides a clean separation between the communication aspect of smart contracts on a blockchain, allowing for the rich interaction patterns, and a programming component, which enjoys principled semantics and is amenable to formal verification. Scilla is not meant to be a high-level programming language, and we are going to use it as a translation target for high-level languages, such as Solidity, for performing program analysis and verification, before further compilation to an executable bytecode.

We describe the automata-based model of Scilla, present its programming component and show how contracts definitions in terms of automata streamline the process of mechanised verification of their safety and temporal properties.

1 Introduction

Smart contracts are a mechanism for expressing computations on a blockchain, i.e., a decentralised Byzantine-fault-tolerant distributed ledger. In addition to typical state of computations, a blockchain stores a mapping from accounts (public keys or addresses) to quantities of tokens owned by said accounts. Execution of an arbitrary program aka a smart contract is done by miners, who run the computations and maintain the distributed ledger in exchange for a combination of gas (transaction fees based on the execution length, denominated in the intrinsic tokens and paid by the account calling the smart contract) and block rewards (inflationary issuance of fresh tokens by the underlying protocol). One distinguishing ability for smart contracts, not found in standard computational settings, is the transfer of tokens between accounts.

One of the challenges of writing smart contracts is that the implemented operational semantics of smart contract languages admit rather subtle behaviours that diverge from the “intuitive understanding” of the language in the minds of contract developers. Some of the largest attacks on smart contracts, e.g., the attack on the DAO [30] and Parity wallet [17] contracts, have turned on such divergencies.1 Software development techniques that have proven very effective in other domains such as app development (e.g., “move fast and break things” [4]) have not translated successfully to smart contract development because it is nearly impossible to patch a contract once deployed due to the anonymous Byzantine execution environment of a public blockchain [29]. Moreover, software engineering techniques, such as static and dynamic analysis tools such as Manticore [6], Mythril [8], Oyente [9], Solgraph [13] and have not yet proven to be effective in increasing the reliability of smart contracts.

Formal methods, such as verification and model checking, are an attractive alternative for increasing the reliability of smart contracts [12, 20, 34]. Formal methods can provide precise definitions for operational behaviour, and therefore can illuminate—and hopefully reduce—subtle behaviours. Generally speaking, formal methods can produce more rigorous guarantees about program behaviour: mathematical proofs instead of summaries of accumulated ad-hoc experience. Moreover, formal methods can provide static guarantees, guaranteeing safety and liveness properties before contracts are irrevocably committed to the blockchain.

In order to apply formal methods efficiently in such a new setting reason about smart contracts and enable efficient language-based verification [51], one must weigh several factors:

• Expressivity. There is a trade-off between making a language simpler to understand and making it more expressive. Bitcoin script [3] occupies the “simpler” end of the spectrum: contracts basically specify validity conditions (simple expressions) that must hold before coins can be transferred. Ethereum [54] occupies the “expressive” end of the spectrum, with a Turing-complete instruction set. However, expressivity is not free. Turing-complete languages are more complex to reason about, especially in an automated manner. Moreover, infinite computations are neither possible nor desirable on a blockchain due to the use of gas to compensate miners (an infinite loop will happily consume as much gas as one cares to feed it even if no progress is being made). It is as-yet unclear if the expressivity of a fully Turing-complete instruction set is necessary to support a practical smart contract ecosystem.

• State. Fundamentally, the blockchain is a stateful database due to the necessity of maintaining and securely updating the mapping between accounts and tokens owned. Moreover, the “event-driven” style of programming employed by many smart contracts (which tend to wait for messages, act on them, and then return to waiting for the next message) requires the storage of contract state between calls. A contract implementing an Initial Coin Offering (ICO) campaign, which records each contributor and the size of their contributions is a standard example of such a stateful event-driven contract. On the other hand, purely functional languages are less error-prone, harder to attack, easier to parallelise, and easier to reason about, so there are good reasons to consider approaches that use mutable state sparingly.

• Communication. Contracts are often used to allow multiple mutually-distrusting parties to interact. This interaction can occur in several ways: by one contract calling another, by raising an event (to be seen and handled off-chain), or by off-chain computations calling back into the blockchain in later blocks. Communication is highly desirable, but can introduce both genuine and faux-concurrent behaviour, especially in a Byzantine environment, enabling attacks due to potentially corrupted state.

1 By sending money to a user-chosen address, the DAO actually called user-chosen code, which in turn executed an unexpected callback into the original contract, which was in a “dirty” state [40].
• **Meaning of execution.** Operational semantics should be clear and principled, minimising the chance of informal misunderstanding. Moreover, there should be support for machine-checked formal reasoning, ideally both automatically-generated (for simpler properties) and human-assisted (for more complex ones).

In this work, we present Scilla: a novel intermediate-level programming language for smart contracts. By “intermediate” we mean that we do not expect most programmers to write in Scilla directly, any more than most programmers write in x86 assembly directly; instead, the typical path will be to compile a higher-level language to Scilla and then further to an executable bytecode, very much in a tradition of optimizing [36] and verified compilers [38]. Scilla aims to achieve both expressivity and tractability, while enabling rigorous formal reasoning about contract behavior, by adopting the following fundamental design principles, based on separation of programming concerns:

Separation between computation and communication. Contracts in Scilla are structured as communicating automata: every in-contract computation (e.g., changing its balance or computing a value of a function) is implemented as a standalone, atomic transition, i.e., without involving any other parties. Whenever such involvement is required (e.g., for transferring control to another party), a transition would end, with an explicit communication, by means of sending and receiving messages. The automata-based structure makes it possible to disentangle the contract-specific effects (i.e., transitions) from blockchain-wide interactions (i.e., sending/receiving funds and messages), thus providing a clean reasoning mechanism about contract composition and invariants.

Separation between effectful and pure computations. Any in-contract computation happening within a transition has to terminate, and have a predictable effect on the state of the contract and the execution. In order to achieve it, we draw inspiration from functional programming with effects, drawing a distinction between pure expressions (e.g., expressions with primitive data types and maps), impure local state manipulations (i.e., reading/writing into contract fields) and blockchain reflection (e.g., reading current block number). By carefully designing semantics of interaction between pure and impure language aspects, we ensure a number of foundational properties about contract transitions, such as progress and type preservation, while also making them amenable to interactive and/or automatic verification with standalone tools.

Separation between invocation and continuation. Structuring contracts as communicating automata provides a computational model, known as continuation-passing style (CPS), in which every call to an external function (i.e., another contract) can be done as the absolutely last instruction. While this programming style is helpful for avoiding multiple pitfalls stemming from interaction between separate contracts (e.g., uncontrolled reentrancy [30]), it might be difficult to program with or use as an intermediate representation. To regain the expressivity, while retaining the principled structure of an automata, we add a special kind of transitions—continuations—that are invoked by the execution environment. Thanks to the mechanism of explicit continuations [47], we can provide a straightforward translation from languages like Solidity to Scilla, yet, keeping the automata structure as a foundational model for analysis and verification.

**Paper outline**

In the rest of this manuscript, we will describe main components of Scilla. In Section 2 we present its computational model, based on communicating automata. In Section 3 we demonstrate the support for reasoning about properties of contract executions, enabled by the automata model. Future design choices w.r.t. contract verification are discussed in Section 4. We provide a survey of related contract language design proposals in Section 5 and conclude in Section 6.

## 2 Contracts as Communicating Automata

In this section, we explain the key concept of Scilla language design using a characteristic example—a crowdfunding campaign à la Kickstarter. In a crowdfunding campaign, a project owner wishes to raise funds through donations from the community. In the specific example modelled here, we assume that the owner wishes to run the campaign for a certain pre-determined period of time. The owner also wishes to raise a minimum amount of funds without which the project can not be started. The campaign is deemed successful if the owner can raise the minimum goal. In case the campaign is unsuccessful, the donations are returned to the project backers who contributed during the campaign.

An implementation of the contract in Scilla is given in Figure 1. The design of the `Crowdfunding` contract is intentionally simplistic (for example, it does not allow the backer to change the amount of donation), yet it shows the important features of Scilla, which we elaborate upon.

The contract is parameterised with three values that will remain immutable during its lifetime (lines 2–4): an owner account address `owner` of type `address`, a maximal block number `max_block` (of type `uint`), indicating a deadline, after which no more donations will be accepted from backers, and a goal (also of type `uint`) indicating the amount of funds the owner plans to raise. The goal is not a hard cap but rather the minimum amount that the owner wishes to raise.

What follows is the block of mutable field declarations (lines 7–10). The mutable fields of the contract are the mapping `backers` (of type `address` ⇒ `uint`), which will be used to keep track of the incoming donations and is initialised with an empty map literal [], and a mutable boolean flag `funded` that indicates whether the owner has already transferred the funds after the end of the campaign (initialised with `false`). In addition to these fields, any contract in Scilla has an implicitly declared mutable field `balance` (initialised upon the contract’s creation), which keeps the amount of funds held by the contract. This field can be freely read within the implementation, as we will demonstrate below, but can only modified by explicitly transferring funds to other accounts.

### 2.1 Transitions and messages

The logic of the contract is implemented by three transitions: `Donate`, `GetFunds`, and `Claim`. The first one serves for donating funds to a campaign by external backers; the second allows the owner to transfer the funds to its account once the campaign is ended and the goal is reached; the final one makes it possible for the backers to reclaim their funds in the case the campaign was not successful.

One can think of transitions as methods or functions in Solidity contracts. What makes them different from functions, though, is the atomicity of the computation enforced at the language level. Specifically, each transition manipulates only with the state of the

---

2https://www.kickstarter.com
contract Crowdfunding
  (owner : address,
  max_block : uint,
  goal : uint)

function max_block
  return uint(1);

function goal
  return 100;

function funded : boolean = false;

transition Donate
  (sender : address, value : uint, tag : string)
  (* Simple filter identifying this transition *)
  if tag == "donate" =>
    blk ← & block_number;
    let nxt_block = blk + 1 in
    if max_block ≤ nxt_block
      then send (<to → sender, amount → 0,
                  tag == "main","msg → "deadline_passed">, MT)
    else send (<to → sender, amount → 0,
                tag == "main", msg → "already_donated">, MT)
  else if (contains(bs, sender))
    then let bs1 = put(bs, sender, value) in
      backers := bs1;
      send (<to → sender, amount → 0,
            tag == "main", msg → "already_donated">, MT)

function GetFunds
  (sender : address, value : uint, tag : string)
  (* Only the owner can get the money back *)
  if (tag == "getfunds") && (sender == owner)
    then blk ← & block_number;
    bal ← & balance;
    if max_block < blk
      then if goal ≤ bal
          then funded := true;
  then if tag == "getfunds"
    then send (<to → owner, amount → bal,
                tag → "main", msg → "funded"), MT)
  else send (<to → owner, amount → 0,
             tag → "main", msg → "failed"), MT)

function Claim
  (sender : address, value : uint, tag : string)
  (* Transition 3: Reclaim funds by a backer *)
  if tag == "claim"
    then bal ← & balance;
    if not (contains(bs, sender)) || funded ||
      goal ≤ bal
      then send (<to → sender, amount → 0,
                 tag → "main", msg → "cannot_refund"), MT)
    else let v = get(bs, sender) in
      backers := remove(bs, sender);
      send (<to → sender, amount → v,
            tag → "main", msg → "here_is_your_money"), MT)

Figure 1. Crowdfunding contract in Scilla: state and transitions.

Every transition in a contract can be roughly thought of as a function that maps an incoming message and an initial contract state to a new contract state and a set of outgoing messages. Ignoring the state-manipulation aspect for a moment, let us consider the functional component of Scilla.

As implementations of transitions in Figure 1 demonstrate, the language syntax includes binding of pure expressions (such as arithmetic and boolean operations, as well as manipulation with mappings) via OCaml-style let-in construct (e.g., lines 19 and 69). Basic control flow also includes branching if-then-else statements, whose semantics is standard. At the moment, we keep an agnostic view w.r.t. the pure component of the language, which will be fixed later and can be as expressive as a polymorphic lambda-calculus [32, 46]. Furthermore, looping constructs are not present in our working example, but we are planning to support them via well-founded recursive function definitions, so their termination can be proved statically.

Every transition’s last command, in each of the execution branches, is either sending a set of messages, or simply returning. Messages are encoded as vectors <...> of name → value entries, including at least the destination address (to), an amount of funds transferred (amount) and a default tag of the function to be invoked (tag). All transitions of the Crowdfunding end by sending a message to either the sender of the initial request or the contract’s owner. For example, depending on the state of the contract and the blockchain, the transition GetFund might end up in either sending a message with its balance to the contract’s owner, if the campaign has succeeded and the deadline has passed, or zero funds with a corresponding text otherwise.

In addition to a message, the trailing send command of a transition includes a continuation value to indicate possible further
execution in a return-flow. The Crowdfunding contract does not require any such executions, so all continuations are “empty” (i.e., they do not initiate any further execution after the callee contract returns), which is indicated by the literal MT. We will provide examples of contracts involving non-trivial return-flow (and, hence, featuring interesting continuations) in Section 2.4.

2.3 State and effects

In addition to performing computations with the components of the incoming messages and parameters of the contract, every transition can manipulate with the state of a contract itself, i.e., read/write from/to its mutable fields, as well as read from the blockchain.

The state of the contract, represented by its fields, is mutable: it can be changed by the contract’s transitions. A body of a transition can read from the fields, assigning the result to immutable stack variables using the specialised syntax \( x \leftarrow \& f \);, where \( f \) is a field name and \( x \) is a fresh name of a local variable (e.g., lines 17 and 42). In a similar vein, a body of transition can store a result of a pure expression \( e \) into a contract field \( f \) using the syntax \( f := e \); (as in lines 27 and 70). The dichotomy between pure expressions (coming with corresponding binding form \( \text{let-in} \)) and impure (“effectful”) commands manipulating the field values, is introduced on purpose to facilitate logic-based verification of contracts, reasoning about the effect of a transition to the contract’s state, while abstracting away from evaluation of pure expressions, similarly to how it is done in functional languages, such as Haskell.

In addition to reading/writing contract state, each transition implementation can use read-only introspection on the current state of the blockchain using the “deep read” operation \( x \leftarrow \& \& g \);, where \( g \) is a name of the corresponding aspect of the underlying blockchain state. For example, the Crowdfunding contract reads the number of a current block in lines 18 and 41. A syntactic emphasis on the contract’s interaction with the blockchain’s current state makes it possible to enable reasoning about contract liveness properties, spanning its long-term behavior, as we will show in Section 3.

At the moment, the model of SCILLA does not feature explicit exceptions, as those are going to be implemented at the level of runtime, without a possibility of raising in the code.\(^\text{3}\)

2.4 Advanced control flow and continuations

It is not uncommon for a contract to call another contract, for instance, implementing a library, and then use the result of the call in the rest of the execution. Currently, the main model of computation in SCILLA prevents this, as it corresponds to a tail-call program form: passing control to another contract can be done only by explicitly sending it a message, not via a call within the transition. Existing high-level languages for contracts, such as Solidity, allow non-tail calls from the middle of a contract execution, and require the notion of program stack in order to handle the returned result. The presence of non-tail calls in Solidity is what enabled the infamous DAO exploit [30], which was due to the fact that the rest of a contract computation, setting the fields accounting for the balance of a contributor, was performed only upon returning from a call to another contract, not before. This led to tail-call programming being advocated as a “good programming practice” for smart contracts written in Solidity-like languages [24].

While the core language of SCILLA enforces non-tail calls from contract transitions, we acknowledge the need for them in certain applications, and introduce an additional component to the execution: explicit continuations. Continuations can be thought of as “the rest of computation”, to be invoked after execution of a call to an external function, being passed the result of the latter. One can also encode exception handling via continuations, which, in that case, would play the role of catch-clauses.

Functional programming languages, such as Haskell and OCaml allow for encoding continuations as closures (first-class anonymous functions), thus implementing a style of programming, known as Continuation-Passing Style (CPS) [19]. In programming languages without first-class functions, such as Solidity, continuations still can be encoded via a dedicated data type that “enumerates” all possible shapes of “remaining computations” and a helper function that gives them an operational meaning, i.e., “executes” the continuation. A transformation of a program from a non-tail-call form to CPS to a form with continuations encoded as a data type (known as a defunctionalisation [28, 47]) is a well-studied topic in the research area of compiler implementation [27], and we adopt it in the design of SCILLA.

Specifically, in Figure 1, every tail call made via send also takes a second argument, continuation MT. This is a constant, “empty”, continuation (hence the name), which indicates that no remaining computation should take place after the execution of a callee contract. However, instead of empty continuation, we could have specified, for instance, a continuation that expects a callee contract result to return a result of type uint, and sends it back in a message to the owner of a caller contract:

```plaintext
(+ Specifying a continuation in a Caller contract +)
continuation UseResult (res : uint)
    send (<to owner, amount -> 0,
        tag -> "main", msg -> res>, MT)

(+ Using a continuation in a transition of Caller +)
transition ClientTransition
    (sender : address, value : uint, tag : string)
    (+ code of the transition +)
    send (<to sender, amount -> 0,
        tag -> "main", msg -> res>, UseResult)

(+ Returning a result in a callee contract +)
transition ServerTransition
    (sender : address, value : uint, tag : string)
    (+ code of the transition +)
    return value
```

That is, in a SCILLA contract, a continuation is very similar to a transition, except it takes not a message, but a value, which is, in turn, returned by a callee contract using a return command, which is an alternative to send and does not send a message. The uniformity of contract-specified continuations and transitions makes it possible to reason about them in the same way as about transitions, in the style demonstrated further in Section 3.

What makes continuations different from transitions is that they are “passive”, i.e., their invocation is handled by the semantics of the execution environment, which maintains them in a stack, invoking the topmost continuation after the current contract executes a transition until the return statement. In contrast, transitions are...
We present a simplified version of Scilla’s semantics in Coq. Coq’s programming component, Gallina, is an ML-family language [31] with a similar syntax, featuring first-class functions, algebraic data types, and records. All data types are immutable, and so are function parameters and local variables, bound via let keyword. Gallina is intentionally non-Turing complete: it does not have an implicit state or pointers; all loops are provably terminating and can be expressed via well-founded recursive functions.

Before translating our first Scilla contract to Coq/Gallina, we first present the structure of the contract encoding, defined in terms of the automata model. Our embedding of Scilla to Coq is, thus, shallow [31]: we model the semantics of the pure (i.e., non message-passing) component of Scilla contracts via native Gallina functions, while the message-passing, state-transition aspect of Scilla is accounted for by encoding contract semantics as a relation in Coq. In essence, this is similar to implementing a regular domain-specific language with a tailored execution runtime on top of Lisp or Scala.

The top part of Figure 2 shows Coq definitions of the main concepts constituting Scilla-like contracts. Three data types represented by Coq’s Structure definitions: message, cstate, and bstate define structures for modelling contract messages, contract states and blockchain state. In our model, messages carry four fields: value, i.e., some amount of funds (isomorphic for now to a natural number of type nat), and address of the message sender analogous to Ethereum’s sender field, a tag indicating a transition of a callee contract to be invoked, and a body message modelling application-specific payload, containing all remaining fields of the message. Generic contract state data type cstate, parameterised by an application-specific state data type $S$, additionally contains two mandatory fields, my_id and balance, storing the contract’s
address (immutable) and a current balance, which might change during the contract’s lifetime. All additional fields are encapsulated in the state component, which is to be defined by the contract implementor. Finally, the bstate data type is used to model the blockchain component that a contract can read from. For simplicity, the figure omits all components but the current latest block number, which is represented by a natural number. Indeed, in the future we are planning to enrich this state for reasoning about contracts that reflect on the blockchain state, e.g., read from the results of previous transactions or even the state of other contracts.

The data type transition describes contract transitions: its ttag component serves to uniquely identify a transition and implement the message-based dispatch (using method field of an incoming message), while tfun implements the transition function, which takes as arguments the contract’s own address, its current balance of type value, state, as well as an incoming message and a current blockchain snapshot, and returns a new state and an optional message. No returned message corresponds to an execution step resulting in an exception.

The contract data type Contract serves to package together contract’s transitions, its address, balance, and initial state, and can be thought of as a template for contract definitions in Solidity-like languages. Most of the fields of the Contract record (parameterised with the application-specific contract state type S) are, thus, self-explanatory. We represent the fixed collection of transitions of a contract automata by a sequence of values of type transition, i.e., tagged transfer functions. The only unusual component of a contract definition is its last property field. It intentionally has no name, which is indicated by the placeholder _. What is important, however, is its type, which asserts a statement. Specifically, it requires that all transitions tags are unique. Intuitively, this is a basic contract validity property, which our encoding enforces at the level of the framework, rather than individual contract. Unlike Solidity and other object-oriented languages, such as Java and C#, taking Coq as a host for domain-specific embedding makes it possible to statically encode and enforce data structure invariants. In other words, it will be impossible to construct a contract in our embedding, such that it has two or more transitions with the same tag. By design, for now this is the only property imposed for any SCILIA contract and verified by a compiler during the embedding into Coq. Any other correctness guarantee is contract specific, and will have to be proven for a particular instance of the Contract data type, i.e., for a particular user-defined contract.

### 3.2 Semantics, safety, and consistency properties

Having defined the basic terminology of contract embedding into Coq, we can now define the meaning of a contract’s behaviour in the form of execution traces. Figure 3 first describes a data type step, which captures a triple, corresponding to single step of a contract taking a particular transition and modifying its state accordingly: a contract pre-state pre, post-state pos and an optional output out. A trace of a contract is defined as a sequence of steps. The auxiliary data type schedule, represented by a sequence of blockchain states and messages, allows us to model all possible interactions between the contract and its environment (i.e., other contracts and the constantly changing blockchain).

To see how all contract traces can be modelled by considering all possible schedules, let us first take a look at the semantic function step_prot that takes a contract state pre, a blockchain state bc and an incoming message m, resulting in a step instance. The way it is defined, it simply finds an appropriate transition in a contract definition C and applies it using the function apply_transition, whose definition we have omitted for brevity. In the case if no transition matching m’s tag is found, the contract’s state and balance are left unchanged; otherwise the new state s’ and an output are obtained and, together with an updated contract balance bal’ are used to construct the final state post.

That is, for a fixed contract C, every component of the schedule (i.e., a pair blockchain state, message) determines its next step, and, hence, the changes in its state and balance. Therefore, given an initial state pre and a schedule sc, we can define a contract execution as a trace, obtained by consecutively processing all components of the schedule by the contract—precisely what is defined by a recursive semantic function execute. Finally, given an initial contract state, balance, and account, the valid execution of a schedule sc is defined via the semantic meta-function execute0, which executes the entire given schedule sc or simply performs an identity step from the initial state in the case if sc is empty.

A trace-based semantics of contracts, provided by means of the definitions of execute and execute0 makes it possible to formulate...
generic classes of contract correctness conditions, independently of the specifics of the end-used contracts or properties of interest.

**Safety.** We first define a predicate \( I \) on a contract state (denoted, in Coq terms, by a “function type” \( \text{cstate} \ S \rightarrow \text{Prop} \) from the type of states \( \text{cstate} \ S \) to propositions \( \text{Prop} \)) to be a safety property if it holds at any state of a contract, that can be obtained as a result of interaction between the contract and its environment, starting from the initial state. The following Coq definition states this formally:

\[
\text{Definition safe} \ (I : \text{cstate} \ S \rightarrow \text{Prop}) : \text{Prop} := \\
(\forall \text{sc} \ \text{pre} \ \text{post} \ \text{out} \ . \ (\forall \text{bc} \ \text{m} \ . \ (\text{method} \ m \in \text{tags} \ p) \rightarrow I \text{pre} \rightarrow I \text{post}) \rightarrow I \text{pre} \land I \text{post})
\]

A safety property means some universally true correctness condition holds at any contract’s state, which is reachable from its initial configuration via any schedule \( \text{sc} \). Typical examples of safety properties of interest include: “a contract’s balance is always positive”, “a contract’s balance equals the sum of balances of its contributors”, or “at any moment no money is blocked on the contract”. The definition above, thus defines safety by universally quantifying over all schedules \( \text{sc} \), as well as step-triples \( \text{Step} \ \text{pre} \ \text{post} \ \text{out} \) that occur in a trace, obtained by following \( \text{sc} \). While this definition is descriptive, it is not very pleasant to work with. This is why we define a natural proof principle to establish safety of a property \( I \). The proof principle is stated as the following Coq’s Lemma:

\[
\text{Lemma safe_ind} \ (I : \text{cstate} \ S \rightarrow \text{Prop}) : \text{Prop} := \\
(\forall \text{sc} \ \text{pre} \ \text{post} \ \text{out} \ . \ (\forall \text{bc} \ \text{m} \ . \ (\text{method} \ m \in \text{tags} \ p) \rightarrow I \text{pre} \rightarrow I \text{post}) \rightarrow I \text{pre} \land I \text{post})
\]

That is, in order to show that the property \( I \) is indeed a safety one, one has to show that (a) it holds in the initial contract’s state \( \text{state}0 \), and (b) if it holds in a pre-state \( \text{pre} \) (i.e., \( I \text{pre} \)), and a contract makes a transition in a blockchain state \( \text{bc} \) via a message \( \text{m} \), then \( I \) holds over the post-state \( I \text{post} \).

The proof of this lemma, justifying the validity of this induction principle, is in our Coq sources and is omitted from this document for the sake of brevity. One can notice that the lemma \( \text{safe_ind} \) is strikingly similar to the classical notion of mathematical induction for natural numbers, and it is indeed, an induction on a number, namely, the length of a schedule and, correspondingly a contract execution trace. This is not the only possible induction principle one can adopt for proving safety of a contract with respect to a given property. For instance, certain other properties are easier to be proven by induction on the size of a certain contract field, considering a trace arbitrary, but fixed. Fortunately, almost any valid proof principle for safety can be encoded in Coq as a lemma, similar to the one above. Our plans involve developing a library of multiple proof principles for proving diverse contract properties, formulated as Coq lemmas, along with the comprehensive documentation on their usage and applicability, so the contract developer could pick one depending on her goal and on the nature of the property or a contract.

**Temporal properties.** Sometimes a property of interest cannot be expressed in terms of a predicate over a single state, as it describes entire sub-traces of a contract execution. Such properties are traditionally expressed using connectives of Temporal Logic [45], that relate two or more states in a trace, generated by executing a state-transition system, such as Scilla contract.

Reasoning principles and the corresponding connectives customary for temporal logic can be encoded in Coq by means of defining logical higher-order operators on traces, in the spirit of higher-order functions in programming languages such as OCaml or Haskell. We are still in the process of determining a minimal set of such connectives, necessary to declaratively describe the contract behaviour. As an example, let us consider a temporal connective \( \text{since_as_long} \ p \ q \ r \), which means the following: once the contract is in a state \( \text{st} \), in which (i) the property \( p \) is satisfied, each state \( \text{st}' \) reachable from \( \text{st} \) (ii) satisfies a binary property \( q \ \text{st} \ \text{st}' \) (with respect to \( \text{st} \)), as long as (iii) every element of the schedule \( \text{sc} \), “leading” from \( \text{st} \) to \( \text{st}' \) satisfies a predicate \( r \).

The corresponding Coq encoding of the \( \text{since_as_long} \) connective is given below. We first specify reachability between states \( \text{st} \) and \( \text{st}' \) via a schedule \( \text{sc} \) as the state \( \text{st}' \) being the last post-state in a trace obtained by executing the contract from \( \text{st} \) via \( \text{sc} \):

\[
\text{Definition reachable} \ (\text{st} \text{st}' : \text{cstate} \ S) \text{sc} := \\
(\text{st}' = \text{post} \ (\text{last} \ (\text{Step} \ \text{st} \text{st}' \text{None} \ (\text{execute} \ \text{sc})))
\]

We next employ the definition of reachability to define the since connective, which is parameterised by predicates \( p, q \) and \( r \). The premises (i)–(iii) are outlined in the following Coq code:

\[
\text{Definition since_as_long} \ (p : \text{cstate} \ S \rightarrow \text{Prop}) := \\
(\text{q : cstate} \ S \rightarrow \text{cstate} \ S \rightarrow \text{Prop}) \\
(r : \text{bstate} \rightarrow \text{message} \rightarrow \text{Prop}) := \\
(\forall \text{sc} \ \text{st} \text{st}' \ . \\
\ (
\text{(i) st satisfies p} \\
\text{p st} \rightarrow \text{(ii) st' is reachable from st via sc} \\
\text{reachable st st' sc} \rightarrow \\
\text{(iii) any element b of sc satisfies r} \\
\ (\forall \text{b, b} \in \text{sc} \rightarrow r \ b) \rightarrow \\
\text{(conclusion) q holds over st and st'} \\
\ q \text{ st'}
\)
\]

Why this logical connective is useful for reasoning about contract correctness? As we will show further, it makes it possible to concisely express “preservation” properties relating contract balance and state, so that they hold as long as certain actions do not get triggered by some of the contract’s users.

### 3.3 Embedding Scilla into Coq

Figure 4 shows the translation of the Scilla implementation of the Crowdfunding contract from Figure 1 to Coq. The translation is mostly straightforward, and for now has been done by hand, but in the future we intend to automate it. We only outline a few discrepancies between the Scilla code and its Coq counterpart,
which were introduced to streamline the reasoning and make full use of Coq’s programming component Gallina.

The state of the contract is defined using Coq’s Structure declaration, familiar from the previous sections. Unlike the code in Figure 1, the definition does not distinguish between immutable contract parameters and mutable fields. However, while we provide getters for all five components of the state (e.g., get_owner, get_backers, etc), we provide setters only for the two last fields, i.e., set_backers and set_funded, which are considered mutable. For now, we model mappings by associative sequences, hence the “field”, backers with an empty sequence (line 30 of Figure 4), and sets the boolean funded flag to false.

The contract’s owner, maximal block determining the end of the campaign, and the funding goal are expressed via Gallina’s Parameters and can be instantiated later, once a specific contract instance is created. Having those three parameters abstract, we define the constructor init_state for the initial state that also instantiates the constructor

```
Definition init_state := CS (init_owner, init_block, init_max_goal) [].
```

(* Initial state *)

```coq
Definition DonateFun := fun id bal s m bc
let bs := backers s in
if (method m == donate_tag) \&\& (from == get_owner s) then
let from := sender m in
let bs := backers s in
if n < size bs
then let bs := contains(bs, sender) bs in
(* new backer *)
let bs' := (from, val m) :: bs in
let s' := set_backers bs' in
(s', Some (Msg 0 id from 0 msg))
else (s, None).
```

### Figure 4. Crowdfunding contract translated into Coq: contract-specific state, initial parameters, and transitions.

as follows. Reading from a contract’s state `x ←& f;` is translated into `let x := get_f s in ...`. Writing into a contract field `f := e;` is translated into `let s' := set_f s e in ...`, where `s'` is the new, modified state to pass further in the computation. Finally, reading from the blockchain `x ←& g;` is encoded via `let x := get_g bc;`. A specific shape of a blockchain getter depends on the property being read. For instance, the current block number can be obtained via `block_num bc`, as shown on line 24 of Figure 4. Following the shallow embedding style, in our translation, we have also formulated SCILLA’s transition filters via Gallina’s native `if-then-else` construct.

Other difference between SCILLA representation and Coq translation involves representing mappings: instead of using language-provided primitives `put/get/contains`, we implement them using Coq’s functions for sequence manipulation. For instance, the call to `contains(bs, sender)` from Figure 1’s line 25 is implemented in Gallina translation via `all [pred e | e.1 != from] bs` in line 28 of Figure 4, checking that no single entry `e` has `from` as its first component, that is, making sure that the sequence does not yet contain a record of `sender m`. In a similar spirit, `put` is encoded via appending a head to a sequence (line 30 of Figure 4), and removing an entry is done via `filter` function (line 66 of Figure 4). Relying on Coq’s support for sequence for encoding mappings makes it possible to reuse a rich library of lemmas about them, thus, sparing us the expense of having to implement a new library for mappings.\(^6\)

\(^6\) Although we might consider implementing such a library in the future to streamline the translation.
For simplicity, at the moment our encoding does not account for explicit reentrancy, i.e., it does not involve continuations, and every execution branch of every transitions terminates by providing a new state (which can be equal to the previous state), as well as an optional output message, to be sent to its destination (third argument of the Msg constructor). The transitions resulting in None as an output should be interpreted as resulting in an exception, which is customary in the functional programming tradition. In this case, the state of the contract is left unchanged.

3.4 Reasoning about contract behaviour

A definition of a contract as a state-transition system and modelling all its execution traces makes it possible to verify its complex properties in isolation, i.e., modularly and independently of other contracts and users that might interact with it. This is achieved by defining safety and temporal properties as universally quantified over schedules, which serve as an external oracle and thus account for all non-determinism of distributed on-chain interaction. Proving a property for all schedules means proving it for any potential adversary, as all its interaction with the contract are limited by what the contract allows for, in terms of its transitions.

We now show how the combination of notions of safety and temporal properties presented above allows us to verify a contract, proving that all its behaviours satisfy a certain complex interaction scenario. Specifically, for our Crowdfunding example, let us prove that, once a donor has been made by a backer with an account address b, given that the campaign eventually fails, the backer b will always be able to get their donation back. There are indeed multiple ways to express this property formally in terms of contract behaviours. For simplicity, we break the statement of interest into three independent components, which, in conjunction provide the requirement stated above.

Property 1: The contract does not leak funds unless the campaign has been funded. First, let us state and prove that the contract does not spend the money, given to it by the backers, unless the campaign has been funded. To do so, we will make use of Coq’s native functions map, used to apply a function to all elements of a sequence, and sum for summing up contents of a sequence. The property of interest, dubbed balance_backed, is as follows:

```coq
definition balance_backed st : Prop :=
  (* If the campaign has not been funded... *)
  funded (state st) ->
  (* the contract has enough funds to reimburse all. *)
  sum (map snd (backers (state st))) = balance st.
```

For an arbitrary contract state st, it asserts that if the funded flag is still false in st (i.e., ¬funded (state st)), then the balance of the contract (balance st) is at least as large as the sum of all donations made by the recorded backers (sum (map snd (backers (state st))).

This property holds for every state of an arbitrary instance of the Crowdfunding contract. To show this, we state the following theorem, claiming that balance_backed is a safety property:

```coq
theorem sufficient_funds_safe : safe balance_backed.
```

The statement of the theorem relies on the definition of safe, instantiating it (implicitly) for an instance of the Crowdfunding contract.

Property 2: The contract preserves records of individual donations. So far, we have proved that the non-funded Crowdfunding contract does not lose money, but what about individual backers? What if the contract takes the donations and silently transfers them from one backer to another, or, even worse, removes the backer from its records, "pretending", that no donation has ever been made. To assert that this is not the case, we rely on the temporal connectivity since_as_long defined above and state that, once a backer made a donation, the record of it is not going to be lost by the contract, as long as the backer makes no attempt to withdraw its donation.

We first define two auxiliary predicates, specific to our contract and the shape of its state:

```coq
(* Contribution d of a backer b is recorded in the field 'backers'. *)
definition donated b (d : value) st :=
  get (backers (state st)), b == [:: (b, d)].

(* b doesn't claim its funding back *)
definition no_claims_from b (q : bstate * message) :=
  sender q.2 != b.
```

The predicate donated specifies that the backer-recording field of the contract has the corresponding entry (b, d). The predicate no_claims_from is defined on schedule bits and ensures that a given schedule component q does not contain a message from the address b. Together, we use these two predicates to state the desired temporal property of the contract:

```coq
theorem donation_preserved (b : address) (d : value):
  since_as_long c (donated b d)
  (fun _ s => donated b d s')
  (no_claims_from b).
```

The mechanised proof of this fact is by induction on the trace connecting the initial state s, in which a donation record is observed, and any fixed subsequent state s’ which is reachable from s via schedules, satisfying the predicate no_claims_from. The existence of such a proof indeed validates the claim that the contract does not mess up with records during its execution, unless the backer tries to claim their donations back.

Property 3: If the campaign fails, the backers can get their refund. By now we know that the contract does not lose the donated funds and keeps the backer records intact. Now we need the last piece: the proof that if a contract is not funded, and the campaign has failed (deadline has passed and the goal has not been reached), then any backer with the corresponding record can get the donation back.

We state the property of interest as theorem can_claim_back in Figure 5. As its premises (a)–(d), the theorem lists all the assumptions about the state of the contract that are necessary for getting
Theorem can_claim_back id b d st bc:
(* (a) The backer b has donated d, so the contract holds
that record in its state *)
donated b d st →
(* (b) The campaign has not been funded. *)
¬ funded (state st) →
(* (c) Balance is small: not reached the goal. *)
balance st < (get_goal (state st)) →
(* (d) Block number exceeds the deadline. *)
get_max_block (state st) < block_num bc →
(* (conclusion) Backer b can get their donation back. *)
∃ (m : message),
  sender m == b ∧
  out (step_prot c st bc m) = Some (Msg d id b 0 ok_msg).

Figure 5. A backer can claim back her funds if the campaign fails.

the reimbursement. The conclusion is somewhat somewhat peculiar: it
expresses the possibility to claim back the funds by postulating the
existence of a message m, such that it can be sent by a backer b,
and the response will be a message with precisely d funds in it,
sent back to b. The theorem, whose proof is only 10 lines of Coq,
formulates the property as one single-step, yet its statement can
be easily shown to be a safety property, as it is, indeed, preserved
by the transitions, and, after the funds are successfully claimed for
the first time, the premise (a) of the statement is going to be false,
hence the property will trivially hold.

Putting it all together. Together, Properties 1–3 deliver the de-
sired correctness condition of a contract: once donated money can be claimed back in the case of a failed campaign. It is indeed not
the only notion of correctness that intuitively should hold over this
particular contract, and by proving it we did not ensure that the
contract is “bug-free”. For instance, in our study we focused on
backers only, while another legit concern would be to formally verify that the contract’s owner will be able transfer the cumulative
donation to their account in the case if the campaign is successful.

In a similar vein, another desired property would be to ensure
that no funds will be locked on the contract forever as recently
happened with the Parity multi-signature wallet [17]: in a finite
amount of time either all backers or the owner will be able to
relieve the contract of its funds. The latter correctness condition is
of particular interest, as it falls into the class of liveness properties,
state that “something good eventually happen”. While we do not
show its proof here, our formalisation of contract executions makes
it possible to prove liveness properties.

4 Discussion
Representing smart contracts as communicating automata enables
multiple opportunities for logic-based verification. It also opens new
challenges with respect to the design of better high-level contract
languages and their expressivity.

4.1 Towards logic-based reasoning about contracts
As we have demonstrated, together, the combination of universal
safety properties and multiple temporal properties (including liveness, which we briefly sketch later in this section) constitute various
contract correctness criteria. Indeed, the notion of correctness is in

the eyes of the beholder: each contract comes with a unique set of
correctness conditions, and there is no single property that captures
all possible notions of “a contract not going wrong”. Even more so, a
behaviour considered erroneous in one contract (e.g., reentrancy in
Ethereum-style contracts) might be a feature exercised consciously
and without harm in another implementation. Defining precisely
the desired contract properties is an art of formal specification and
is outside the scope of this technical presentation. We believe that
having a clean contract semantics and an expressive logical abstrac-
tions is of crucial importance for formally describing the behaviour
of blockchain-based applications.

Therefore, we do consider our mission to provide semantic foundations,
as well as a logical vocabulary for making it possible to for-
mulate and prove all reasonable contract properties, giving Scilla
programmers a toolset to implement, specify, and mechanically verify
the contracts with respect to correctness conditions of interest.

We also believe that prototyping a contract language by encoding
its programs and their semantics in a state-of-the art language with
dependent types and a support for formal machine-assisted proofs,
such as Coq, Agda [21], F* [52], or Idris [22] provides a principled
way to rigorously specify and verify the implementation for a large
class of notions of correctness in terms of the programs themselves,
rather than their models [50].

4.2 On Turing-completeness and contract language design
Our host framework Coq’s programming language Gallina, which
was used to implement a verifiable version of Scilla by means of
shallow embedding, is not Turing-complete: each well-typed Coq
function, when applied to concrete arguments, will terminate in
a finite number of steps. By defining Scilla and its embedding to
Coq in a way that its pure and state-manipulating in-transition
computations are modelled by Coq’s pure total functions, we en-
sured that all Scilla transitions terminate. This made it possible
to conduct verification by means of symbolic execution of Coq’s
expressions with explicit state. Keeping the programming compo-
nent of transfer functions strictly terminating, in the future, we
should be able to provide better support for automating the proofs of
safety/temporal properties by employing third-party tools, such as
TLA+ [37] and Ivy [4]. In our proof-of-concept contract ver-
ification effort, we have not explicitly accounted for analysis of
resources or computational effects, such as exceptions. Such effects
can be modelled in a similar way, i.e., by explicit encoding of an
effect-passing and cost-counting discipline, or by engineering a
version of an indexed monad to keep track of the effect in the type
system [16, 42].

While transitions of Scilla contracts are strictly terminating by
construction, they still might take arbitrarily long to compute, e.g.,
in the presence of a large data value passed by a message. Even
more, one is still able to implement potentially non-terminating (in
the absence of limited stack length or gas bounds) computations by
making a contract calling itself, directly or indirectly, via a message-
passing mechanism or explicit continuations. This is similar to
implementing recursion via so-called “Landin’s knot”, i.e., not by
means of the language function calling mechanisms, but through
the (blockchain) state, storing a computation (i.e., a contract). To
be able to statically detect such scenarios before a transaction is
fired, we are going to implement an analysis to deliver precise
parameterised cost boundaries on transition executions.
5 Related Proposals

Every blockchain either has a (limited) scripting language for transaction validation as in say Bitcoin or a general purpose smart contract language as in Ethereum. In the recent years, several new languages improving upon languages in Bitcoin and Ethereum have been proposed by the community. Each language is usually designed in the context of a specific underlying blockchain. Below, we compare Scilla and some of the existing smart contract languages and the improvement proposals.

- **Typecoin (Bitcoin)** [26]: Typecoin is a logical commitment mechanism built on top of Bitcoin to carry logical propositions. The underlying idea in Typecoin is to have a transaction carry logical propositions instead of coins. In fact, each Bitcoin transaction can be translated into a Typecoin transaction where the inputs and outputs become propositions and the logic would allow to split or merge inputs. This allows transactions to be type-checked before they get committed to the blockchain. Since, UTXOs can only be merged or split, the underlying logic is linear in nature and is not rich to handle complex states as in Scilla.

- **Simplicity (Bitcoin)** [43]: Simplicity is the most recent language proposed in the context of Bitcoin. Since Bitcoin uses a UTXO model and the state of the system is not as complex as in say Ethereum, the language design does not need to handle read and write to a global state. As a result, the language follows Bitcoin’s design of self-contained transactions whereby contracts do not have access to any information outside the transaction. This further implies that Simplicity does not support communicating contracts. Scilla on the other hand manages read and write to a shared memory space and is designed for an account-based model, where contracts can communicate with each other.

- **Solidity (Ethereum)** [14]: Solidity is the most popular smart contract language today. It is a Turing complete language and resembles JavaScript. However, the expressivity of the language has introduced avenues for several vulnerabilities in the past. For instance, a class of re-entrancy attacks was shown to be possible due to arbitrary interleaving of local state manipulations and external calls [30]. Scilla restricts the computation model to communicating automata and mandates external calls to occur at the end of the transition. Also, CPS style explicit passing of return values to the caller makes reasoning about programs much easier. With Scilla, we further show how to prove critical safety and liveness properties on the contract. As for Solidity, the Turing completeness nature of the language makes contracts less amenable to formal verification.

- **Bamboo (Ethereum)** [2]: Among all language proposals, Bamboo is the closest to Scilla. Bamboo relies on polymorphic contracts where one contract changes to another whenever a state change occurs. While, in Scilla the transition function changes the state while the transition itself does not change. In fact, Bamboo’s morphing of contracts can be easily encoded via a state component in Scilla. The other difference being that Bamboo does not focus on the verification aspect of the contracts.

- **Babbage (Ethereum)** [1]: Babbage is a conceptual-level design of a smart contract proposed in the context of Ethereum. The design adopts a mechanical model of writing contracts as opposed to a textual program. But, due to its underlying simplicity and lack of formal design semantics, it is hard to compare it with Scilla and assess its amenability to formal reasoning.

- **F* embedding (Ethereum)**: Bhargavan et al. [20] provide a framework to analyze and verify both the runtime safety and the functional correctness of Ethereum contracts written in Solidity by translation to F*, a function programming language aimed at program verification. The work however does not present a new programming model as in Scilla.

- **Viper (Ethereum)** [15]: Viper is an experimental language proposed in the context of Ethereum to ease the auditability of smart contracts. The language does not have recursive calling and infinite loops. As a result, one can eliminate the need to set an upper bound on gas limits that is known to be vulnerable to gas limit attacks. Viper also plans to remove the possibility of making changes to the state after any non-static calls. The idea being that this will prevent reentrancy attacks. Scilla takes a slightly stricter approach where any external call has to be the last instruction of a transition function.

- **Rholang (RChain)** [11]: While Scilla is based on communicating automata, Rholang is based on asynchronous polyadic π-calculus that is best suited to work in a concurrent setting. Rholang admits unbounded recursion, while Scilla will only allow bounded recursion. The other difference comes from the fact that Scilla mandates all external calls to be tail calls. As a result, there is no complex interleaving of external calls and local instructions. Hence, analyzing and proving safety properties in Scilla becomes much easier than in Rholang.

- **Michelson (Tezos)** [7]: Michelson is a purely functional stack-based language that has no side-effects. On the other hand, Scilla is not purely functional as transitions do affect the external state.

- **Liquidity (Tezos)** [5]: Liquidity is a high-level language that complies with the security restrictions of Michelson. Because of its compatibility with Michelson, a similar comparison with Scilla holds.

- **Plutus (IOHK)** [10]: Plutus is a language based on typed λ-calculus designed to run transaction validation for a blockchain like Bitcoin. As a result, the language is simpler that Scilla, which allows contracts to manipulate global state and make external calls to other contracts. Also, because of the automata based design, Scilla contracts are easily composable.

- **OWL (BOSCoin)** [23]: OWL for Web Ontology Language is being developed in the context of BOSCoin. The underlying computation model in OWL is Timed Automata [18], and hence bears some similarity with Scilla. However, OWL advocates for pure functions without side effects, while Scilla is not purely functional and hence allows complex yet cleanly separated interactions with other contracts.

- **F* dialect (Zen)** [40]: Zen uses a dialect of F* for its smart contract language. The smart contracts in Zen are stateless and are functionally pure. This means that there are no side effects and no interaction with other contracts. While, this removes race conditions and any barriers to parallel execution it also severely limits the kind of smart contracts that one can build. For instance, multiple transactions involving the same smart contract may not be easily parallelised, and may have to be executed in series. In Scilla, we understand the need of impure functions and we also understand the complexities arising from it. As a result, Scilla’s computation model cleanly separates local computations and external calls that require communication. With a clean separation, it becomes possible to eliminate complex and undesirable interleaving of local computations and external calls.
6 Conclusion & Future Work

In this work, we outlined the design of Scilla—an intermediate level language for smart contracts. Scilla provides a clear separation between the communication aspect of a smart contract and its programming component. The underlying computation model in Scilla is based on communicating automata. We also presented an embedding of a Scilla contract to Coq and proved safety and liveness properties of the contract. Future work consists in defining a formal grammar and semantics of the language and implementing Scilla, as well as a developing and verifying a number of contracts in it, on a real-world blockchain platform.

Acknowledgments: This research was partially supported by a grant from the National Research Foundation, Prime Minister’s Office, Singapore under its National Cybersecurity R&D Program (TSUNAmi project, No. NRF2014NCR-NCR001-21) and administered by the National Cybersecurity R&D Directorate. This research was partially supported by a grant from Yale-NUS College R-607-265-322-121.

References