Scilla: a Smart Contract Intermediate-Level LAnguage

Automata for Smart Contract Implementation and Verification

Ilya Sergey
University College London
i.sergey@ucl.ac.uk

Amrit Kumar
National University of Singapore
amrit@comp.nus.edu.sg

Aquinas Hobor
Yale-NUS College
National University of Singapore
hobor@comp.nus.edu.sg

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 contract 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 property of 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 behaviour 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 [29] 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] 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, 33]. Formal methods can provide precise definitions for operational behaviour, and therefore can illuminate and hopefully reduce subtle behaviour. 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 to 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 [48].

- **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 [35] and verified compilers [37]. 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 this, 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 [29]), 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 *wrt.* 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.[2] 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 backers to change the amount of their 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

[2] https://www.kickstarter.com

```
1   contract Crowdfunding
2    (owner     : address,
3     max_block : uint,
4     goal      : uint)
5
6   (* Mutable state description *)
7   {
8     backers : address ⇒ uint = [];
9     funded  : boolean = false;
10  }
11
12  (* Transition 1: Donating money *)
13  transition Donate
14    (sender : address, value : uint, tag : string)
15    (* Simple filter identifying this transition *)
16    if tag == "donate" ⇒
17    bs ⟵ & backers;
18    blk ⟵ && block_number;
19    let nxt_block = blk + 1 in
20    if max_block ≤ nxt_block
21    then send (<to ⟶ sender, amount ⟶ 0,
22                tag ⟶ "main",
23                msg ⟶ "deadline_passed">, MT)
24    else
25      if not (contains(bs, sender))
26      then let bs1 = put(bs, sender, value) in
27           backers := bs1;
28           send (<to ⟶ sender,
29                  amount ⟶ 0,
30                  tag ⟶ "main",
31                  msg ⟶ "ok">, MT)
32      else send (<to ⟶ sender,
33                  amount ⟶ 0,
34                  tag ⟶ "main",
35                  msg ⟶ "already_donated">, MT)
36  (* Transition 2: Sending the funds to the owner *)
37  transition GetFunds
38    (sender : address, value : uint, tag : string)
39    (* Only the owner can get the money back *)
40    if (tag == "getfunds") && (sender == owner) ⇒
41    blk ⟵ && block_number;
42    bal ⟵ & balance;
43    if max_block < blk
44    then if goal ≤ bal
45         then funded := true;
46              send (<to ⟶ owner, amount ⟶ bal,
47                     tag ⟶ "main", msg ⟶ "funded">, MT)
48         else send (<to ⟶ owner, amount ⟶ 0,
49                     tag ⟶ "main", msg ⟶ "failed">, MT)
50    else send (<to ⟶ owner, amount ⟶ 0, tag ⟶ "main",
51                msg ⟶ "too_early_to_claim_funds">, MT)
52
53  (* Transition 3: Reclaim funds by a backer *)
54  transition Claim
55    (sender : address, value : uint, tag : string)
56    if tag == "claim" ⇒
57    blk ⟵ && block_number;
58    if blk ≤ max_block
59    then send (<to ⟶ sender, amount ⟶ 0, tag ⟶ "main",
60                msg ⟶ "too_early_to_reclaim">, MT)
61    else bs ⟵ & backers;
62         bal ⟵ & balance;
63         if (not (contains(bs, sender))) || funded ||
64            goal ≤ bal
65         then send (<to ⟶ sender, amount ⟶ 0,
66                     tag ⟶ "main",
67                     msg ⟶ "cannot_refund">, MT)
68         else
69         let v = get(bs, sender) in
70         backers := remove(bs, sender);
71         send (<to ⟶ sender, amount ⟶ v, tag ⟶ "main",
72                msg ⟶ "here_is_your_money">, MT)
```

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

contract itself, without involving any other contracts or parties. All interaction with the external world, with respect to the contract, happens either at the very start of a transition, when it is initiated by an external message, or at the end, when a message (or messages), possibly carrying some amount of funds, can be emitted and sent to other parties.

Each transition can be invoked by a suitable message, which should provide a corresponding *tag* as its component to identify which transition is triggered. It is enforced at the compile time that tags define transitions unambiguously. All other components of the message, relevant for the transition to be executed, are declared as the transition's parameters. For instance, the transition `Donate` expects the incoming message to have at least the fields `sender`, `value`, and `tag`. What follows in each transition's definition is the *filter*—an optional clause **if** e ⇒, where e is a boolean-returning computation that can involve reading from the components of the incoming message and the contract's state, deciding whether the corresponding transition can be taken. For instance, the transitions `Donate` and `Claim` only check that the tag of a message matches that of the transition; the filter of `GetFunds` (line 40) additionally checks that the sender of the message is the contract's owner.

To keep the logic of filters simple, we deliberately disallow complex expressions in them, as well as write-interaction with the contract's state. Any further checks relating to the incoming message with the contract's state can be implemented in a transition's body, as we describe further.

2.2 Basics of program flow

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 *wrt.* the pure component of the language, which will be fixed later and can be as expressive as a polymorphic lambda-calculus [31, 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 ←& 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 **`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 ←&& 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.[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 [29], 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:

```
(* 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 its 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

[3] This design choice might change in the future, in favor of implementing exceptions as another computational effect.

"active", *i.e.*, they should be explicitly invoked, externally or by other contracts, by sending a message.

Notice that a continuation can itself "schedule" another continuation for later execution as a rest of computation. For instance, in the code snippet above the `UseResult` continuation ends with sending a message and scheduling an `MT` continuation. In principle, a continuation can even schedule itself, which would mean a potentially non-terminating execution. That is, the only non-well-founded recursion in Scilla can be implemented via contracts calling themselves or other contracts in a circular way. Such potentially non-terminating computations involving multiple transitions and continuations are going to be handled using the *gas* mechanism, similar to the one implemented in Ethereum. In contrast, standalone transitions of a contract *always* terminate.

In the future, we are going to implement an automatic translator from a subset of Solidity into Scilla by employing CPS transformation and defunctionalisation, in order to produce Scilla contracts formulated in terms of transitions for direct control flow and continuations for return-flow.

3 Mechanised Verification of Scilla Contracts

We now turn our attention to *formally reasoning* about executable contracts implemented in Scilla. The language's automata-based model allows us to state a formal semantics of a contract's executions, both independent and parameterised via interaction scenarios with other contracts, as well as to rigorously capture the properties of a contract's executions during its life-cycle. Below, we show how to reason both about safety (*nothing goes wrong*) and liveness (*certain things may eventually happen*) properties.

We are developing Scilla hand-in-hand with the formalisation of its semantics and its embedding into the Coq proof assistant [25]—a state-of-the art tool for mechanised proofs about programs, based on advanced dependently-typed theory and featuring a large set of mathematical libraries.[4] In the past, Coq has been successfully applied for implementing certified (*i.e.*, fully mechanically verified) compilers [37], operational systems [32], concurrent [49] and distributed applications [38], including blockchains [44].

Further in this section, we will show the translation from Scilla to Coq (which is mostly straightforward), as well as the definition of contract protocols, semantics and safety/liveness properties, along with the corresponding proof machinery. We will then present and discuss a series of properties of the `Crowdfunding` contract from Section 2 that we have verified. In this manuscript, we outline a preliminary simplified model of contracts, which does not feature resource semantics (*i.e.*, Ethereum-style "gas"), full-fledged blockchain reflection, and advanced control-flow features, such as continuations and exceptions. All these aspects are orthogonal to the automata-based model we are describing; that can and will be modelled in the future in our framework for formal verification, by enhancing the Coq model of the Scilla contracts and its semantics.

3.1 Contracts in Coq: basic definitions and properties

We present a simplified version of Scilla's semantics in Coq. Coq's programming component, Gallina, is an ML-family language [40] 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.

```
Structure message :=
  Msg { val : value;
         sender : address;
         method : tag;
         body : payload }.
Structure cstate (S : Type) :=
  CState { my_id : address;
            balance : value; state : S }.
Structure bstate :=
  BState { block_num : nat;
            (* More components of the blockchain. *) }.

Structure transition :=
  CTrans { ttag : tag;
            tfun : address → value → S → message →
                   bstate → (S * option message); }.

Structure Contract (S : Type) := Contr {
      (*Account id *)
      acc : address;
      (* Initial balance *)
      init_bal : nat;
      (* Initial state of the contract protocol *)
      init_state : S;
      (* Contract comes with a set of transitions *)
      transitions : seq (transition S);
      (* All transitions have unique tags *)
      _ : uniq (map ttag transitions) }.
```

Figure 2. Basic definitions: states, transitions, and contracts.

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* [30]: 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

[4] The mechanised embedding of a subset of Scilla into Coq is publicly available for downloads and experiments: https://github.com/ilyasergey/scilla-coq.

(immutable) address 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 snapshots, 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 Scilla 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).

```
(* A single-step execution: pre/post states and output;
   contract-specific state S is now assumed to be fixed. *)
Structure step :=
  Step { pre : cstate S;
         post : cstate S;
         out : option message }.
Definition trace := seq step.
Definition schedule := seq (bstate * message).

(* In the following definition, a contract automata C
   is implicit and fixed. *)
Definition step_prot (pre : cstate S) (bc : bstate)
                     (m : message) : step :=
  let CState id bal s := pre in
  let (s', out) := apply_transition C id bal s m bc in
  let bal' := if out is Some m'
              then (bal + val m) - val m' else bal in
  let post := CState id bal' s' in
  Step pre post out.

(* Map a schedule into a trace *)
Fixpoint execute (pre : cstate S) (sc: schedule) : trace :=
  if sc is (bc, m) :: sc'
  then let stp := step_prot pre bc m in
       stp :: execute (post stp) sc'
  else [::].

Definition state0 :=
  CState (acc C) (init_bal C) (init_state C).
Definition execute0 sc :=
  if sc is _ :: _ then execute state0 sc
  else [:: Step state0 state0 None].
```

Figure 3. Contract traces and semantics.

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" `cstate S →Prop` from the type of states `cstate S` to propositions `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:

```
Definition safe (I : cstate S → Prop) : Prop :=
  (* For any schedule sc, pre/post states and out... *)
  ∀ sc pre post out,
  (* such that the triple Step (pre, post, out)
     is in the trace obtained via sc *)
  Step pre post out ∈ execute0 sc →
  (* both pre and post satisfy I *)
  I pre ∧ I 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 `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 `sc`, as well as step-triples `Step pre post out` that occur in a trace, obtained by following `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`:[5]

```
Lemma safe_ind (I : cstate S → Prop) :
  (* (a) *)
  I state0 →
  (* (b) *)
  (∀ pre bc m, (method m ∈ tags p) → I pre →
               I (post (step_prot pre bc m))) →
  (* (conclusion) *)
  safe I.
```

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 `state0`, *and* (b) if it holds in a pre-state `pre` (*i.e.*, `I pre`), and a contract makes a transition in a blockchain state `bc` via a message `m`, then `I` holds over the post-state: `I (post (step_prot pre bc m))`.

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 `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 `since_as_long p q r`, which means the following: once the contract is in a state `st`, in which (*i*) the property `p` is satisfied, each state `st'` reachable from `st` (*ii*) satisfies a binary property `q st st'` (with respect to `st`), as long as (*iii*) every element of the schedule `sc`, "leading" from `st` to `st'` satisfies a predicate `r`.

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

```
Definition reachable (st st' : cstate S) sc :=
  st' = post (last (Step st st None) (execute st 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 corresponding comments in the following Coq code:

```
(* q holds since p, as long as schedule bits satisfy r. *)
Definition since_as_long (p : cstate S → Prop)
                         (q : cstate S → cstate S → Prop)
                         (r : bstate * message → Prop) :=
  ∀ sc st st',
    (* (i) st satisfies p *)
    p st →
    (* (ii) st' is reachable from st via sc *)
    reachable st st' sc →
    (* (iii) any element b of sc satisfies r *)
    (∀ b, b ∈ sc → r b) →
    (* (conclusion) q holds over st and st' *)
    q st 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,

[5] One can think of lemmas as of "library functions", whose statement (type) is of importance but the proof (implementation) is opaque and can be ignored by the users.

```
1   (* Contract-specific state S *)
2   Structure crowdState := CS {
3     owner : address;
4     max_block : nat;
5     goal : value;
6     backers : seq (address * value);
7     funded : bool }.
8
9   (* Initial parameters *)
10  Parameter init_owner : address.
11  Parameter init_block : nat.
12  Parameter init_goal : value.
13
14  (* Initial state *)
15  Definition init_state :=
16    CS (init_owner, init_block, init_max_goal) [::] false.
17
18  (* Transition: tag and a transfer function. *)
19  Definition donate_tag := 1.
20
21  Definition donate_fun := fun id bal s m bc ⇒
22    if method m == donate_tag then
23      let bs := backers s in
24      let nxt_block := block_num bc + 1 in
25      let from := sender m in
26      if get_max_block s <= nxt_block
27      then (s, Some (Msg 0 id from 0 no_msg))
28      else if all [pred e | e.1 != from] bs
29           (* new backer *)
30           then let bs' := (from, val m) :: bs in
31                let s' := set_backers s bs' in
32                (s', Some (Msg 0 id from 0 ok_msg))
33           else (s, Some (Msg 0 id from 0 no_msg))
34    else (s, None).
35
36  Definition donate := CTrans donate_tag donate_fun.
```

```
37  Definition getfunds_tag := 2.
38  Definition getfunds_fun : tft := fun id bal s m bc ⇒
39    let from := sender m in
40    if (method m == getfunds_tag) && (from == get_owner s) then
41      let blk := block_num bc + 1 in
42      if get_max_block s < blk
43      then if get_goal s <= bal
44           then let s' := set_funded s true in
45                (s', Some (Msg bal id from 0 ok_msg))
46           else (s, Some (Msg 0 id from 0 no_msg))
47      else (s, Some (Msg 0 id from 0 no_msg))
48    else (s, None).
49  Definition get_funds := CTrans getfunds_tag getfunds_fun.
50
51  Definition claim_tag := 3.
52  Definition claim_fun := fun id bal s m bc ⇒
53    let from := sender m in
54    if method m == claim_tag then
55      let blk := block_num bc in
56      if blk <= get_max_block s
57      then (* Too early to ask for reimbursements! *)
58        (s, Some (Msg 0 id from 0 no_msg))
59      else let bs := backers s in
60           if (funded s) || (get_goal s <= bal)
61           (* Cannot reimburse: campaign succeeded *)
62           then (s, Some (Msg 0 id from 0 no_msg))
63           else let n := seq.find [pred e | e.1 == from] bs in
64                if n < size bs
65                then let v := nth 0 (map snd bs) n in
66                     let bs' := filter [pred e | e.1 != from] bs in
67                     let s' := set_backers s bs' in
68                     (s', Some (Msg v id from 0 ok_msg))
69                else (* Didn't back or have already claimed *)
70                  (s, None)
71    else (s, None).
72  Definition claim := CTrans claim_tag claim_fun.
```

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

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` is encoded as a sequence of pairs `(address, value)`, indicating the backers and the corresponding amount they have donated.

The contract's *owner*, maximal block determining the end of the campaign, and the funding goal are expressed via Gallina's `Parameter`s 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 backers with an empty sequence `[::]`, and sets the boolean `funded` flag to `false`.

The remaining Coq code defines the encoding of the three transitions of the contract, by means of specifying their tags (*e.g.*, `donate_tag`), transfer functions (*e.g.*, `donate_fun`), and packaging them together in a single transition (*e.g.*, `donate`). In this version of the encoding, we model implicit mutable state of the contract by explicit functional *state-passing style*. That is, each transition's transfer function takes the current contract's state `s`, as well as the incoming message `m` and a blockchain state `bc` as its parameters. The mapping between Scilla primitive commands is then 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 passed 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 [53]. 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* [34] 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 donation `d` has been made by a backer with an account address `b`, given that the campaign eventually fails, the backer `b` will be always able to get their donation `d` 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 `sumn` for summing up contents of a sequence. The property of interest, dubbed `balance_backed`, is as follows:

```
Definition balance_backed st : Prop :=
  (* If the campaign has not been funded... *)
  ¬ funded (state st) →
  (* the contract has enough funds to reimburse all. *)
  sumn (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 (`sumn (map snd (backers (state st)))`).

Does this property hold for every state of an arbitrary instance of the `Crowdfunding` contract? To show this, we state the following theorem,[7] claiming that `balance_backed` is a safety property:

```
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, as well as (explicitly) for the property `balance_backed`. A machine-checked proof of the theorem, which takes less than 50 lines of Coq code, is conducted by using the induction principle `safe_ind` defined above. For instance, the property `balance_backed` clearly holds in the initial state of the contract, as the set of backers is empty (`[::]`), and the balance cannot be negative. Thus, we can formally prove that a non-funded campaign does not lose/spend money of its backers.

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 connective `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:

```
(* 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:

```
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

[7] `Theorem` declarations in Coq are no different from `Lemma`s, but are usually considered more important.

```
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 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 desired 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, stating 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 abstractions is of crucial importance for formally describing the behaviour of blockchain-based applications.

Therefore, we consider our mission to provide semantic foundations, as well as a logical vocabulary for making it possible to formulate 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 ensured 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 component 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+ [36] and Ivy [43]. In our proof-of-concept contract verification 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, 41].

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)** [42]: 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 [29]. 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.
- **$\mathsf{F}^\star$ 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 $\mathsf{F}^\star$, 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.
- **$\mathsf{F}^\star$ dialect (Zen)** [39]: Zen uses a dialect of $\mathsf{F}^\star$ 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. Hobor's research was partially supported by a grant from Yale-NUS College R-607-265-322-121. Sergey's research was partially supported by EPSRC First Grant EP/P009271/1.

References

[1] Babbage—a mechanical smart contract language. https://medium.com/@chriseth/babbage-a-mechanical-smart-contract-language-5c8329ec5a0e.

[2] Bamboo. https://github.com/pirapira/bamboo, accessed on Nov 30, 2017.

[3] Bitcoin Script. https://en.bitcoin.it/wiki/Script, accessed on Dec 2, 2017.

[4] Common Vulnerabilities and Exposures. https://cve.mitre.org/index.html, accessed on Dec 2, 2017.

[5] Liquidity. http://www.liquidity-lang.org/.

[6] Manticore. https://github.com/trailofbits/manticore, accessed on Dec 2, 2017.

[7] Michelson: the language of Smart Contracts in Tezos. https://www.tezos.com/static/papers/language.pdf.

[8] Mythril. https://github.com/b-mueller/mythril/, accessed on Dec 2, 2017.

[9] Oyente. https://github.com/melonproject/oyente, accessed on Dec 2, 2017.

[10] Formal Specification of the Plutus Core Language (rev. 10). https://github.com/input-output-hk/plutus-prototype.

[11] Rholang. https://rholang.rchain.coop/, accessed on Nov 30, 2017.

[12] Securify. https://securify.ch/, accessed on Dec 2, 2017.

[13] Solgraph, . https://github.com/raineorshine/solgraph, accessed on Dec 2, 2017.

[14] Solidity, . https://solidity.readthedocs.io/en/develop/, accessed on Nov 30, 2017.

[15] Viper. https://viper.readthedocs.io/en/latest/.

[16] Danel Ahman, Catalin Hritcu, Kenji Maillard, Guido Martinez, Gordon D. Plotkin, Jonathan Protzenko, Aseem Rastogi, and Nikhil Swamy. Dijkstra monads for free. In *POPL*, pages 515–529. ACM, 2017.

[17] JD Alois. Ethereum Parity Hack May Impact ETH 500,000 or $146 Million, 2017. https://www.crowdfundinsider.com/2017/11/124200-ethereum-parity-hack-may-impact-eth-500000-146-million/, accessed on Dec 2, 2017.

[18] Marcin Andrychowicz, Stefan Dziembowski, Daniel Malinowski, and Lukasz Mazurek. Modeling bitcoin contracts by timed automata. In *FORMATS*, volume 8711 of *LNCS*, pages 7–22. Springer, 2014.

[19] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[20] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal Verification of Smart Contracts: Short Paper. In *PLAS*, pages 91–96. ACM, 2016.

[21] Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In *TPHOLs*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

[22] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[23] Yezune Choi and Jake Hyunduk Choi. Owlchain(BOScoin) Technical Specification.

[24] ConsenSys. Smart contract security best practices. https://github.com/ConsenSys/smart-contract-best-practices.

[25] Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.7.0*, 2017. http://coq.inria.fr/.

[26] Karl Crary and Michael J. Sullivan. Peer-to-peer affine commitment using Bitcoin. In *PLDI*, pages 479–488. ACM, 2015.

[27] Olivier Danvy. Defunctionalized interpreters for programming languages. In *ICFP*, pages 131–142. ACM, 2008.

[28] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at Work. In *PPDP*, pages 162–174. ACM, 2001.

[29] Michael del Castillo. The DAO Attacked: Code Issue Leads to $60 Million Ether Theft, 2016. https://www.coindesk.com/dao-attacked-code-issue-leads-60-million-ether-theft/, accessed on Dec 2, 2017.

[30] François Garillot and Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In *TPHOLs*, volume 4732 of *LNCS*, pages 368–382. Springer, 2007.

[31] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[32] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *OSDI*, pages 653–669. USENIX Association, 2016.

[33] Yoichi Hirai. Formal Verification of Ethereum Contracts (Yoichi's attempts). https://github.com/pirapira/ethereum-formal-verification-overview, accessed on Dec 2, 2017.

[34] Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. Oracle Semantics for Concurrent Separation Logic. In *ESOP*, volume 4960 of *LNCS*, pages 353–367. Springer, 2008.

[35] Simon L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[36] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002. ISBN 0-3211-4306-X.

[37] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, pages 42–54. ACM, 2006.

[38] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In *POPL*, pages 357–370. ACM, 2016.

[39] Asher Manning. Zen Protocol's Smart Contract Paradigm. https://blog.zenprotocol.com/zen-protocols-smart-contract-paradigm-a6e54a187d84.

[40] Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1990. ISBN 978-0-262-63132-7.

[41] Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Dependent types for imperative programs. In *ICFP*, pages 229–240. ACM, 2008.

[42] Russell O'Connor. Simplicity: A New Language for Blockchains. https://blockstream.com/simplicity.pdf.

[43] Oded Padon, Kenneth L. McMillan, Aurojit Panda, Mooly Sagiv, and Sharon Shoham. Ivy: safety verification by interactive generalization. In *PLDI*, pages 614–630. ACM, 2016.

[44] George Pîrlea and Ilya Sergey. Mechanising blockchain consensus. In *CPP*, pages 78–90. ACM, 2018.

[45] Amir Pnueli. The Temporal Logic of Programs. In *FOCS*, pages 46–57. IEEE Computer Society, 1977.

[46] John C. Reynolds. Towards a theory of type structure. In *Programming Symposium*, volume 19 of *LNCS*, pages 408–423. Springer, 1974.

[47] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[48] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *WTSC*, volume 10323 of *LNCS*, pages 478–493. Springer, 2017.

[49] Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. Mechanized verification of fine-grained concurrent programs. In *PLDI*, pages 77–87. ACM, 2015.

[50] Ilya Sergey, James R. Wilcox, and Zachary Tatlock. Programming and Proving with Distributed Protocols. *PACMPL*, 2 (POPL):28:1–28:30, 2018.

[51] Tim Sheard, Aaron Stump, and Stephanie Weirich. Language-based verification will change the world. In *FOSER*, pages 343–348. ACM, 2010.

[52] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278. ACM, 2011.

[53] Philip Wadler. How to Replace Failure by a List of Successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA*, volume 201 of *LNCS*, pages 113–128. Springer, 1985.

[54] Gavin Wood. Ethereum: A Secure Decentralized Generalised Transaction Ledger, 2014.