CHAPTER 1

The Next 700 Smart Contract Languages

Smart contracts is a category of special-purpose programs that implement computations, whose executions are replicated by means of a distributed consensus protocol. The parties participating in the consensus protocol can, thus, define custom logic for transactions by deploying smart contracts that implement arbitrary applications to be executed in a decentralised way. Smart contracts have proven to be the most useful in the finance domain and have been extensively used for implementing various forms of digital accounting, voting, and schemas for distribution of rewards.

Smart contracts have attracted a lot of interest from the research community, due to their correctness-critical nature, and also because of a number of costly accidents that involved deployed faulty contract implementations. In the retrospect, many of those issues could have been avoided with a more careful choice of linguistic abstractions, that is, with a programming language design tailored for the domain.

In this chapter, we provide a high-level overview of the design choices in programming languages for smart contracts that are dictated by their essential aspects: (i) ***atomicity***, (ii) ***communication***, (iii) ***management of digital assets***, and (iv) ***resource accounting***. We argue that finding a balancing act for expressing those concepts poses a fundamental challenge in the programming language design for blockchain-based decentralised applications.

1.1 INTRODUCTION

Smart contracts are a mechanism for expressing replicated computations powered by a decentralised consensus protocol [85]. They are most commonly used to define custom logic for transactions operating over a blockchain, *i.e.*, a decentralised Byzantine-fault-tolerant distributed ledger [14, 73]. 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 management of token transfers between accounts. While simple forms of smart contracts were already available for regulating exchange of virtual coins in earlier cryptocurrencies such as Bitcoin [66], smart contracts owe their wide adoption to the Ethereum framework [19, 94]. Since their first public implementations in mid-2010s, protocols supporting smart contract deployment have found many applications in digital finance, accounting, voting, gaming and many other areas that naturally require decentralisation.

One of the challenging design aspects of smart contracts is the fact that the outcomes their executions is determined by their interactions with a decentralised adversarial environment. That is, once deployed, smart contracts might exchange data with other contracts, possibly designed to exploit vulnerabilities in their logic, with the aim to provoke an unforeseen token transfer or execute a denial-of-service attack [58]. Such vulnerabilities are typically caused by rather subtle contract behaviour that diverges from the "intuitive understanding" of the language in the minds of the contract developers. Some of the most prominent attacks on smart contracts deployed on Ethereum blockchain, *e.g.*, the attack on the DAO[1] [30] and Parity wallet [8] contracts, have been exploited some undocumented or poorly understood behaviour of certain language features. These and many others reported attacks on the Ethereum-deployed contracts [13, 51, 55, 56, 67, 76] have made *execution safety and formal correctness guarantees* to be the primary concerns for smart contract-based programming.

The fact that third-party smart contracts cannot be trusted has another important implication on the design of the implementation language semantics. The decentralised nature of contract execution means that the majority of the involved miners will have to agree on the outcome of a transaction involving a contract's invocation. Therefore, by providing the runtime support for contract executions, the miners become open to a wide class of denial-of-service attacks. Such attacks may be effectively implemented by contracts that never terminate under certain conditions or use an extensive amount of memory. As a remedy to this challenge, in Ethereum's pioneering approach to smart contracts every transaction costs a certain amount of *gas* [19, 94], a monetary value Ethereum's currency, paid by a transaction-proposing party. Computations (performed by invoking smart contracts) that require more computational or storage resources, cost more gas than those that require fewer resources. Gas cost is deduced dynamically: each execution step is being charged from the gas supply paid for; if a transaction "runs out of gas" in the midst of its execution, it is interrupted, with all the corresponding changes discarded. Therefore, an

[1] Decentralised Autonomous Organisation [32].

adequate model for principled gas accounting is crucial for the definition of a smart contract language semantics and reasoning about the safety and the dynamic costs of its executions. As the recent research shows, miscalculated gas costs, both at the level of the language and of the individual contracts are quite common [5, 22, 38, 61, 70], and they might lead to even more severe vulnerabilities to exploit.

At the beginning of the smart contract adoption, it was hard to predict what kinds of applications the clients will most likely be willing to develop and deploy. While the encodings of financial routines form the majority of smart contracts to date, Ethereum's design provided a very expressive run-time environment for smart contracts—the Ethereum Virtual Machine (EVM) [94]. EVM offered a Turing-complete low-level language, whose features, amongst others, included arbitrary interaction between contracts (with any contract's code accessible to any other contract), dynamic contract creation, and ability to introspect on the entire state of the Ethereum blockchain. Such versatility made EVM a very popular platform for developing high-level languages to compile to, which in turn resulted in an explosion of Ethereum applications ranging from fully decentralised auctions and fundraisers, crowdfunding to multiplayer games, protocols for verified computations, and even fraud schemes [31, 62, 63]. The *expressivity* and a low-level language design are a double-edged sword. While offering a great flexibility in implementing custom transaction logic, it is frequently at odds with the safety aspect outlined below. For instance, Ethereum contracts, deployed in a low-level language render independent audit and formal verification of deployed code infeasible in practice. Narrowing the scope of the minimal necessary functionality that smart contracts need to possess is, thus, an active ongoing research.

The history of programming language (PL) design and implementation is also a history of making programs run *faster*, by exploiting the corresponding problem domains as well as underlying hardware architectures. In this regard, smart contracts are seemingly no different from other programs, and optimising their runtime will also benefit the entire protocol. However, the above mentioned safety and expressivity aspects, optimisations of smart contracts pose a number of unusual challenges for the language and runtime designers. For instance, it is not clear how the optimisations will interact with the gas costs defined for a particular execution mode. Furthermore, the smart contract runtimes to date to the large extent treat the underlying consensus protocols as a black box, without taking any advantage of its architecture, which might allow for parallel execution of transactions [4, 54, 59]

1.1.1 WHAT WE WILL DISCUSS

The three dimensions of the smart contract language design can, be, thus, summarised by a diagram shown in Figure 1.1. As a reference point, we illustrate

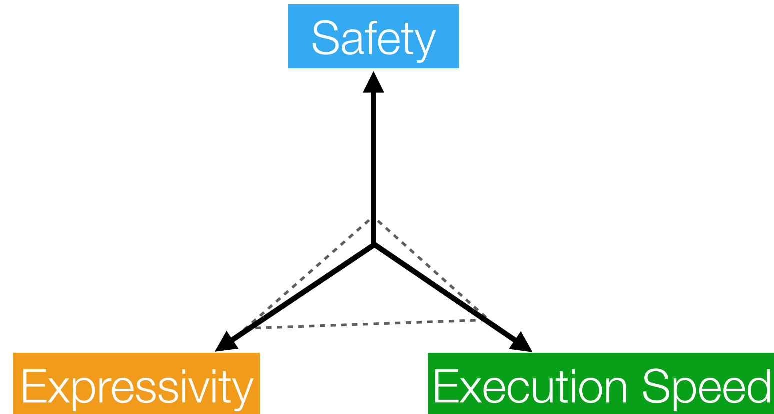

Figure 1.1: Language Design Trade-off. The dashed line shows EVM's design choices.

Ethereum's EVM design choices by a dashed line, emphasising its focus on the expressivity and optimisation-friendliness, but not so much on formal guarantees of execution safety.

In this chapter we aim to provide tentative answers to the following questions:

1. What are the essential concepts of smart contracts that by all means need to be represented in the language used to implement them?
2. Which of the well-established PL techniques can be useful for this task and what are the challenges in adapting them?
3. What are the unsolved problems in smart contract language design that one should consider tackling in the future?

In the rest of this chapter, we will elaborate on these three dimensions, discussing various components specific to smart contract programming, and outlining multiple possibility of programming language abstractions targeting greater expressivity, safety, or providing a more suitable ground for program optimisations.

1.1.2 WHAT WE WILL NOT DISCUSS

At the time of this writing, the smart contract programming landscape is growing at a breakneck speed, with new language proposals emerging nearly every week. To date, most of those languages are available in a form of a sparsely documented repository, a position paper, or a blog post [2, 7, 23, 44, 48, 74, 75]. Therefore, we do not aim to provide a detailed survey of the currently available smart contract programming technology, but rather focus on the conceptual components that either have been or

might have been encompassed in some of those proposals (in which case appropriate examples will be provided).

In the past few years, the efforts on discovery, analysis, modeling, and fixing specific classes of vulnerabilities in Ethereum smart contracts have turned into an active research field [9, 10, 15, 16, 21, 38, 39, 40, 51, 55, 56, 58, 61, 67, 87, 89]. While some of those techniques are informative for PL design for smart contracts, their formulation is, in most of the cases, very specific to the Ethereum platform and EVM. The survey of those approaches and tools is, thus, beyond the scope of this chapter as well. The readers interested in contemporary state of the art in those directions are encouraged to check the corresponding survey papers [11, 79].

1.2 BACKGROUND

To set up the stage for the discussion on smart contract language design, let us first consider a simple smart contract and understand its behaviour and properties. Figure 1.2 shows one of the most common applications of smart contracts—a crowdfunding campaign—implemented in Ethereum's Solidity programming language [34].[2] Solidity is a high-level language with the syntax similar to JavaScript, and it compiles directly to EVM. As of early 2020, this is a de-facto programming language for smart contracts that has received wide adoption due to the popularity of Ethereum protocol.

The contract in Figure 1.2 is very similar to a stateful object in a language such as Java or C#. It features four mutable *fields*. The field `owner` of type `address` defines the identity of the account that deploys the contract. The fields `goal` and `deadline` set the main parameters of the crowdfunding campaign: the amount of currency it is aimed to raise and the deadline (*i.e.*, the final block) upon reaching which the donations are no longer accepted. Finally, the field `backers` of type `mapping (address => uint256)` is a mutable hash-map that stores the amounts donated by different backers identified by their account addresses.

The *constructor* `Crowdfunding` of the contract sets the fields `owner`, `deadline` and `goal` for the values provide upon the contract deployment. The value of the owner is retrieved from the field `sender` of the implicit constructor argument `msg` that denotes the message initiating the interaction with the contract (in this case, its deployment) in the corresponding transaction. Thus, `msg.sender` refers to the account that has initiated the transaction. Upon the successful execution of the constructor the resulted state and the code of the contract are replicated amongst the miner nodes.

All the subsequent interactions with the contract by the third parties are done by means of invoking its `function`s (methods), of which it has three. The first one,

[2] This contract is adopted from `https://programtheblockchain.com/posts/2018/01/19/writing-a-crowdfunding-contract-a-la-kickstarter/`.

```
contract Crowdfunding {
  address owner;
  uint256 deadline;
  uint256 goal;
  mapping(address => uint256) backers;

  function Crowdfunding(uint256 numberOfDays, uint256 _goal) public {
    owner = msg.sender;
    deadline = now + (numberOfDays * 1 days);
    goal = _goal;
  }

  function donate() public payable {
    require(now < deadline);        // before the fundraising deadline

    backers[msg.sender] += msg.value;
  }

  function claimFunds() public {
    require(address(this).balance >= goal); // funding goal met
    require(now >= deadline);         // after the withdrawal period
    require(msg.sender == owner);

    msg.sender.call.value(address(this).balance)();
  }

  function getRefund() public {
    require(address(this).balance < goal);  // campaign failed: goal not met
    require(now >= deadline);         // in the withdrawal period

    uint256 donation = backers[msg.sender];
    backers[msg.sender] = 0;
    msg.sender.call.value(donation)();
  }
}
```

Figure 1.2: A Crowdfunding contract in Solidity.

`donate` allows to transfer the donation to the contract. The first line of the method checks, via the `require` construct, that the current block (referred to as `now`) is strictly smaller than the `deadline`. If this test fails, the whole transaction is reverted. The amount of currency transferred to the contract from the party initiating the interaction is implicitly stored in the attribute `value` of the incoming message. As the function is marked `payable`, this amount will be *implicitly* added to the contract's balance—the code does not contain and instructions to do so. For the purpose of

correct accounting, the method records the donated value attributed to the message sender to the map `backers`.

The purpose of the method `claimFunds` is to allow the owner of the contract to transfer all the funds from the contract to its own account. The method first performs a number of checks, ensuring that the collected balance of the contract is larger or equal than the set `goal`, and that the deadline has passed, as well as the sender of the message is indeed the initial `owner`, the contract performs the transfer of funds to the sender of the message, *i.e.*, the owner via a somewhat unusual Solidity's construct `call.value(...)()`. The final method `getRefund` allows the backers to retrieve their donations, transferring the correct amounts of currencies back to their own accounts, after the deadline has passed and if the goal has not been reached (those conditions are checked in the first place). If this is indeed the case, the donation amount is retrieved from the `backers` map, and then transferred to the backer in the last line of the method. Altogether the three methods of `Crowdfunding` contract form its interface, defining all modes of interaction with it. It is important to notice that, by its nature, all smart contracts are *passive*, *i.e.*, then do not engage into any interactions pro-actively. Instead, the code in their methods is executed as a reactions to the messages sent by the external accounts, which might belong to users or other contracts serving as interaction "proxies".

1.2.1 THE SUBTLETIES OF THE CROWDFUNDING CONTRACT

While seemingly very simple, the `Crowdfunding` contract has a number of intricacies, and misunderstanding of any of those might lead to the deployment of a flawed implementation, which will result in a potential loss of funds.

Consider, for instance, the very first `donate` method. Its only purpose is to check that the current block has a number smaller than the set deadline, and, if it is the case, accept the incoming funds from the backer. The subtle point in this logic is that the map `backers` might already store the previous donation, and, hence, the new donation needs to be added to it. This logic is implemented correctly in line 16 via the `+=` operator. That said, by a very trivial oversight, one could write instead `backers[msg.sender] = donation`. This logical mistake would not be caught by the compiler, yet, it would result in the backer losing its previous donation, which she would not be able to retrieve. How so? Consider the logic of `getRefund` method. The donation amount transferred back to the backer is taken from the `backers` map, which, given the described above mistake, would only store the *latest* donation by the backer! That is, the amount donated previously would be no longer accounted for and would not be returned to the backer. Quite ironically, the described programming mistake would not prevent the owner from cashing out the funding collected in the case of a successful campaign. This is due to the fact that the code in line 24

of Figure 1.2 operates with the contract's `balance` rather than the contents of the `backers` map, thus depleting the entire balance accumulated by the contract as the result of it previously accepting donations.

Another possible bug that would render the `Crowdfunding` contact useless would be if the programmer forgot to put the instruction `backers[msg.sender] = 0` in the line 32. In this case, the backer would be able to get the refund several times, possibly until the contract's balance is depleted.

Swapping lines 32 and 33 would lead to even more interesting behaviour. According to Solidity's semantics, the execution of the command `msg.sender.call.value(donation)()` would transfer not only the funds, but also the control over execution to the account `msg.sender`, thus allowing it to perform some additional operations, before returning the control to the `Crowdfunding` contract. Specifically, the account of `msg.sender` could belong to another contract, which could invoke `getRefund` again, thus withdrawing more funds, similarly to the previous scenario in which we forgot to set the backer's donation to 0. This behaviour is known as *reentrancy vulnerability* [41]. It has been a source of the most famous bug in a contract deployed in Ethereum [30], sparking a line of research targeting to prevent these mistakes in the future [40, 51, 89].[3]

1.2.2 REASONING ABOUT CONTRACT PROPERTIES

Smart contracts are safety-critical applications. While existing static analysis tools help to significantly reduce the risk of deploying a faulty contract [1, 51, 71], when designed for a language as complex as Solidity/EVM they will inevitably be unsound (*i.e.*, might miss some bugs). By designing a language that avoids certain mistakes in programs by construction, one can achieve stronger correctness guarantees.

As demonstrated by the `Crowdfunding` example, despite their simple logic of state manipulation, smart contracts are not always easy to get right. Not only the programmer must keep in mind all the properties relevant to the normal contract behaviour prior to its deployment, she also needs to be aware of the precise semantics of certain language constructs in order to avoid the unexpected outcomes. These complexity is exacerbated by the fact that, once deployed (*i.e.*, replicated via the blockchain), smart contracts cannot be patched or amended.

One way to ensure that the contract obeys some "common sense" before it is deployed is state its properties and ensure that the code preserves them. Those properties are commonly phrased as contract *invariants*, *i.e.*, the assertions that holds at any point of the contract's life time. For instance, the following set of formal properties makes for a reasonably complete specification of the `Crowdfunding` contract:

[3] As the result of this bug, Solidity has been augmented with a number of restrictive primitives for transferring funds, namely `send` and `transfer`, which are preferable in most of the similar scenarios.

P1 (No leaking funds). The contract's accounted funds do not decrease unless the campaign has been funded or the deadline has expired.

P2 (Donation record preservation). The contract preserves cumulative records of individual donations by backers, unless they interact with it.

P3 (The backer can get refunded). If the campaign fails, the backers can eventually get their refund for the whole amount they have donated. They can get this refund exactly once.

The recent works have shown how to formally *prove* these properties using machine-assisted tools [25] for a small smart contract language [81, 82]. However those ironclad correctness guarantees are only available if the *entire language*'s semantics is well-defined and formalised. This semantics should describe, among other execution artefacts, contract deployment and interaction with other entities in the blockchain. As of now, no fully formal semantics of Solidity exists, and this is why existing verification tools [71] for Ethereum have to rely on the *ad-hoc* understanding of the language's runtime.

1.2.3 CONTRACT EXECUTION MODELS

Even though contracts are to the large extent just stateful replicated objects, the choice of the language paradigm for implementing them should not be solely based on this fact. While JavaScript-like syntax and semantics have been chosen for Solidity to simplify its adoption, the same choice makes formal reasoning and efficient compilation far from straightforward. For instance, it has been observed that most of the contracts' functionality in fact falls into the pure functional fragment of data manipulation, with state manipulation comprising only a small part of the contract implementations. This informed the design of several contract languages, such as Michelson [86], Liquidity [68], and Scilla [83].

Another factor determining the choice of programming abstractions is the *state model* that is supported by the underlying blockchain consensus protocol. For instance, Ethereum follows an *Account/Balance* model, wherein all state of the user and the contracts (including funds attributed to them) is stored in a data base-like fashion, with account addresses serving as unique keys. An alternative model, adopted by Bitcoin [66] and Cardano [53] blockchains, is based on *unspent transaction outputs* (UTxO) model [84], wherein transactions form a directed acyclic graphs, threading the state of individual accounts from the initial until the terminal nodes. The UTxO model has been shown a better fit for functional programming model [20].

1.2.4 GAS ACCOUNTING

In the `Crowdfunding` example from Figure 1.2, all methods of the contract are straight-line code, with no loops or recursion. However, those features are frequently necessary for implementing common machinery. For instance, executing an iteration over a list of backers' account addresses would allow to reimburse all of them in a single call instead of calling `getRefund` for each backer individually.

Unfortunately, statically unbounded iteration, as well as general loops and recursion would be a source of denial-of-service attacks on the entire blockchain protocol, as all miners would have to run the code that potentially cannot terminate. As of now, the community has reached a consensus that precisely defined execution cost semantics, *aka gas*, is an inherent part of any smart contract language that is planned to be used in an open system, where arbitrary parties can join. That said, allocating adequate gas costs is far from straightforward and to date this aspect of contract language design has received little attention from the research community. In § 1.6, we will discuss some non-obvious challenges when assigning gas costs to smart contract executions as well as programming language-enabled techniques for analysing gas usage patterns.

1.2.5 ON THE ROLE OF TYPES

Types in programming languages are going to be a paramount motive of our overview [72]. Modern type systems provide a compositional syntactic approach for ensuring strong syntactic guarantees for a variety of execution properties. We will see how type-based approaches help to ensure stronger atomicity guarantees (§ 1.3), restrict communication patterns (§ 1.4), enforce the correct handling of digital assets (§ 1.5), and reason about resource consumption (§ 1.6).

1.3 ENFORCING CONTRACT INVARIANTS

The reentrancy vulnerability is the DAO contract [30, 41] is a great example of violating one of the foremost principles in designing software that might interact with other untrusted components — *preservation of invariants* [80]. Invariants are logica assertions that postulate certain relations between the components of the contract. For instance, the property P2 of the `Crowdfunding` contract is an invariant that can be expressed as the following mathematical assertion:

$$\texttt{now} \geq \texttt{deadline.} \vee \sum_{\texttt{b} \in \mathrm{dom}(\texttt{backers})} \texttt{backers}(\texttt{b}) = \texttt{balance} \tag{1.1}$$

Invariants are a common way to reason about validity of a state of an evolving object at any point of its lifetime, and the invariant-based reasoning is customarily

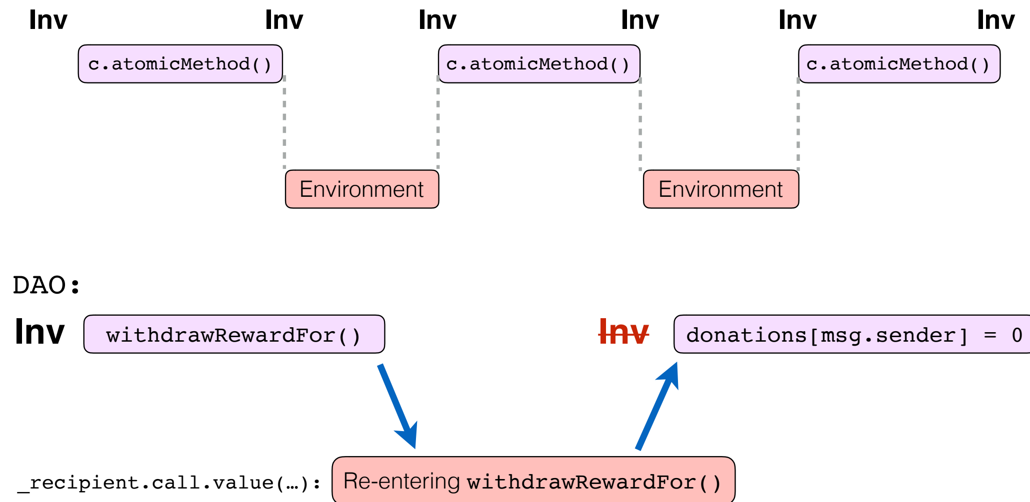

Figure 1.3: Invariant-preserving interactions with an atomic object (top). Invariant-violating re-entrant behaviour in the DAO contract (bottom).

used to argue for the correctness of mutable concurrent data structures [43]. In such objects, each modification in the object's state by a single process should be performed *atomically*, *i.e.*, it cannot be interrupted by other processes in the midst of its execution. A concurrent object's methods are implemented in such a way that they may only violate the object's invariants during their execution, but must restore it at the end of the call. Consider Figure 1.3. Its top part shows an interaction between a concurrent object `c` and its *environment*. Whenever the method `c.atomicMethod()` is executed, it assumes a certain invariant, and, upon termination, it restores it. This way, next time the same method is called, it may still rely on the invariant being preserved. Taking another look at the `Crowdfunding` contract in Figure 1.2, we can see that the invariant (1.1) indeed holds after the contract is initialised by its constructor, and all methods preserve it. One way to look at the contract, thus, is as to a valid *concurrent object*, which maintains its own state invariants.

This principle is, however, violated by the DAO contract, whose reentrant execution is schematically shown in the bottom of Figure 1.3. In the middle of the call to `withdrawRewardFor()` method, a method of another contract is called via `_recipient.call.value(...)`, and, upon returning the execution back to `withdrawRewardFor()` the invariant is not restored. This lack of atomicity made it possible for the adversaries to exploit the "dirty" state of the contract and deplete it of its funds by calling the `withdrawRewardFor()` method again, in a re-entrant way [80].

Issues of this nature could be avoided should the language design would be more restrictive than what's allowed in Solidity. For instance, in Scilla language for smart contracts [83] any *non-atomic* interactions between contracts are forbidden by design. That is, the only way for a contract to call another contract is to first finish its own execution and only then pass the control to the others. While this design does not guarantee that the invariants will be preserved (as those tend to be complex and depend on the domain), it eliminates the DAO-like reentrancy scenario whatsoever.

The downside of by-construction atomicity. The language-enforced atomicity comes with a cost, as it requires the developer to design a contract in a way that makes all its modifications self-contained and not depending on the intermediate interaction with other contracts. This shortcoming does not exclude the possibility to implement most of the typical contract applications, but makes some of them quite cumbersome. A particular class of such applications are *oracles*—services that provide off-chain data to the contracts by means of invoking *callback* methods [80]. Furthermore, the need to make contracts self-contained prevents an efficient code reuse across multiple applications deployed on a blockchain.

Code reuse with pure functions. While state-manipulating code of external contracts cannot be called when one's goal is to enforce atomicity, this restriction can be lifted for *pure* functions. The results of pure computations do not involve mutable state and are obtained as mathematical functions of their inputs. This makes them safe to use in an atomic environment, as their outcomes will not be affected by the effects of other contracts' executions. This insight has been implemented by a number of contemporary smart contract languages [49, 69, 83], which have chosen the design heavily inspired by general-purpose functional programming languages, such as Standard ML and Haskell, in which the pure and state-manipulating computations can be distinguished syntactically or by means of an expressive type system.

Type systems for invariant preservation. Sometimes the contract invariants can be so generic that they might be captured in a form of types. For instance, the post-deadline configuration of the `Crowdfunding` contract can be described as a form of *typestate* [6]—an approach allowing to include state information in types of the variables that can be modified (or not) while the object is in this state. For instance, one can define a typestate PostDeadline, which is the only one, in which it is allowed for the contract o send money to the third parties. Typestate-based approach has been implemented in languages Flint [77, 78] and Obsidian [24] to provide stronger guarantees for contracts implemented in the Solidity style object-based model.

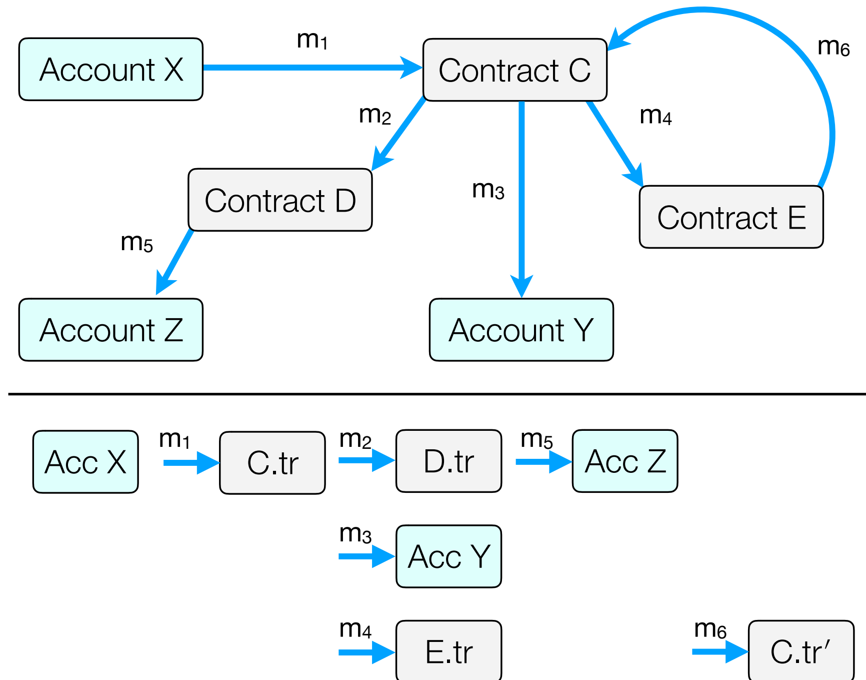

Figure 1.4: An interaction between accounts and contracts within a transaction (top), and its sequentialisation when executed by the protocol (bottom).

1.4 STRUCTURING COMMUNICATION

The need to enforce atomicity of contracts also hints a solution for arranging interactions between them.

Solidity has leaned to the object-based model, familiar to seasonal Java and C# developers, in which all contract interactions are simply method calls. As we've seen above, this model makes it non-trivial to enforce by-construction atomicity and requires additional mental efforts to structure the contract in a way that it would always preserve its invariants.

A more suitable approach for this purpose is to implement interaction protocols between contracts as *communication* via passing messages. The message-passing paradigm for building application of multiple interacting entities is well-studied both from the theoretical and practical perspectives. The most notable theoretical frameworks that describe message-passing programs are π-calculus [64], actors [3] and Input/Output Automata [60]. Practical implementation of those concepts can be found in general-purpose programming languages, such as Erlang [12] and Scala [42]. The

idea of implementing smart contracts as communicating state-transition systems has been first explored in Scilla [81] language developed for Zilliqa blockchain [95]. A contract in Scilla is a definition of mutable and immutable state components (with the former define upon deployment), as well as number of "transition", each serving as a handler reacting to a certain kind of messages. Transitions are atomic, and may result in sending more messages to other contracts, which will be processed later.

A transition invocation may trigger a chain of contract calls as shown in Figure 1.4 (top). In case of a multi-contract transaction (*i.e.*, when a contract interacts with other contracts), the emitted messages are sequentialised by following a *breadth-first* traversal of the transaction communication graph (figure bottom). The messages are then executed in sequence.[4] The combined output of the set of messages resulting from a transaction is committed to the blockchain atomically, in the sense that nothing is committed unless all messages succeed. If one message completes and the next one runs out of gas, the entire transaction is rolled back.

Type systems for message-passing. The idea of implementing contract interactions via explicit message passing and atomic transitions have been implemented in other languages for smart contracts: Rholang [74] and Nomos [28]. Both languages feature an expressive type system that statically enforces certain user-provided interactions between contracts. For instance, Nomos' type system, inspired my resource-aware binary session types [29] ensures that communication between two contracts spanning multiple messages and transitions will follow a particular protocol while also consuming a specified fixed amount of computational resources (*cf.* § 1.6.1).

1.5 IT'S ALL ABOUT MONEY

After all, the main purpose of smart contracts to date is to manage digital assets. Solidity's take on considering money as simply an unsigned integer datatype is prone to make programmers commit errors that would lead to loss of significant amounts of funds. A significant amount of research effort has been dedicated to remedying this design choice with a help of tools for automated analyses for Ethereum contracts. Such tools can check that, for instance, a contract does not transfer its own funds to any third party unconditionally (so-called *prodigal* scenario) [67]. In this section we survey the programming language techniques that allow to ensure the correct handling of digital money by a contract.

Cash-flow analysis. The Scilla smart contract language has introduced a static analysis whose purpose is to ensure that the value representing money are never mixed with other values in an inconsistent way [50]. Specifically, the cash-flow anal-

[4]Breadth-first was chosen over depth-first as it provides better fairness guarantees for message processing.

ysis attempts to determine which parts of the contract's state (*i.e.*, its fields) represent an amount of money, in order to ensure that money is being accounted for in a consistent way. To do so the analysis employs standard techniques of abstract interpretation [26], so each field, parameter, local variable, and subexpression in the contract is given a *tag* indicating if and how it is used *wrt.* representing money.

Running the analysis on the `Crowdfunding` contract (Figure 1.2) results in the fields `goal` and `backers` of the contract being tagged as money-containing. The `goal` field represents the amount of money the owner of the contract is trying to raise, rather than an amount of money owned by the contract. However, the field is still tagged as "money", since its value is regularly compared to the value of `balance`. As the `backers` field is a map, the analysis determines that its values represent money.

In the case if money-related fields are handled inconsistently (*e.g.*, the value of `goal` is added to `deadline`), the analysis will report an error. This treatment of asset-related fields via the static analysis is similar to the idea of inferring *units of measure* as an auxiliary information for types in a program [52].

Type systems for managing digital assets. Static safety of asset management is an excellent application for type-based techniques. A particularly prominent idea is to use *linear types* [37, 91]—a form of a type system that ensures that values that a program operates with are always "consumed" exactly once. Linear types have been previously used to control resource usage in functional programs [90]. As such, linear types are also a good fit to describe the data type of assets. This way, the crucial property of "no double spending" of digital assets will be enforced by the type discipline, as a double spending would be similar to "double usage", which is precisely what linear types prevent from happening.

Linear types were first used in the Typecoin system [27] to ensure the absence of double spends Bitcoin scripts [17]. A similar idea has been used in the Nomos language [28], in which linear types were integrated with session types, and were used to define the consistent money transfer as a part of statically-enforced communication protocols. Both Flint [77] and Obsidian [24] languages define a form of Asset type whose values obey the linearity property. Flint's notion of assets as prevents the values of that type from being duplicated or destroyed accidentally. This choice leads to certain limitations of how assets are handles in Flint: for example they cannot be returned from functions. Obsidian addresses these issues by instrumenting its types with *access permissions*, controlling the ways certain values are used. This way, it permits any non-primitive type to be an asset and treated as a first-class value.

Move [18] is a statically typed stack-based bytecode language with a syntactic layer providing an intermediate representation which is sufficiently high-level to write human-readable code, yet directly translates to Move bytecode. The key feature of Move's type system is the ability to define custom resource types with

semantics inspired by linear logic [37]: a resource can never be copied or implicitly discarded, only moved between program storage locations. Move's programming model assumes having a global mapping of addresses representing the blockchain entities to assets (resources). Therefore, anyone, can change their account by publishing new resources representing "currencies" of different kinds. Linearity of resources ensures that users cannot lose or duplicate their assets when transforming them from one kind to another. Compared to Move, Scilla does not have a notion of a global mutable mapping from addresses to assets. This implies that contract authors commonly have to maintain their own local mappings of addresses to assets tailored to the purpose of the contract at hand. Handling those local mappings can often be simplified by using a generic escrow-like contract published alongside with the specific contract the programmer wants to deploy [88].

1.6 EXECUTION COSTS AND GAS ACCOUNTING

The rationale behind the resource-aware smart contract semantics, instrumented with gas consumption, is three-fold. First, paying for gas at the moment of proposing the transaction does not allow the emitter to waste other parties' *computational power* by requiring them to perform a lot of worthless intensive work. Second, gas fees disincentivise users to consume too much of *replicated storage*, which is a valuable resource in a blockchain-based consensus system. Finally, such a semantics puts a cap on the number of computations that a transaction can execute, hence prevents denial-of-service attacks based on non-terminating executions (which could otherwise, *e.g.*, make all miners loop forever).

The specification of EVM provides a detailed specification of gas cost allocations for all its primitive commands [94], with a lot of focus on costs for interacting with the storage. For instance, EVM's memory model defines three areas where it can store items: the storage is where all contract state variables reside, every contract has its own storage and it is persistent between external function calls (transactions) and quite expensive to use; the memory is used to hold temporary values, and it is erased between transactions and is cheaper to use; the stack is used to carry out operations and it is free to use, but can only hold a limited amount of values.

The gas-aware operational semantics of EVM has introduced novel challenges *wrt.* sound static reasoning about resource consumption, correctness, and security of replicated computations:

1. It is discouraged in the Ethereum safety recommendations that the gas consumption of smart contracts depends on the size of the data it stores (*i.e.*, the contract state), as well as on the size of its functions inputs, or of the current state of the blockchain [35]. However, according to the recent study, almost 10% of the functions of Ethereum contracts feature such a dependency [5]. The inabil-

ity to estimate those dependencies, and the lack of analysis tools, leads to design mistakes, which make a contract unsafe to run or prone to exploits. For instance, a contract whose state size exceeds a certain limit, can be made forever stuck, not being able to perform any operation within a reasonable gas bound. Those vulnerabilities have been recognized before, but only discovered by means of unsound, pattern-based analysis [38].

2. While the EVM specification provides the precise gas consumption of the low-level operations, most of the smart contracts are written in high-level languages, such as Solidity [34] or Vyper [33]. The translation of the high-level language constructs to the low-level ones makes static estimation of runtime gas bounds challenging, and is implemented in an ad-hoc way by state-of-the art compilers, which are only able to give constant gas bounds, or return ∞ otherwise.

1.6.1 CONTROLLING GAS CONSUMPTION WITH PL TECHNIQUES

The programming language research to date has addressed the first challenge by applying a number of techniques to support more accurate gas consumption analysis. One of the best studies approaches for enforcing non-functional properties such as resource consumptions is by means of employing *sub-structural type systems*, allowing to the programmers to declare the desired boundaries on resources consumed by the program and letting the type checker (as a part of compilation pipeline) to ensure statically that these boundaries are respected at run-time [46, 93]. As writing explicit resource boundaries might impose large annotation overhead, thus, slowing down the contract development, type-based techniques are usually combined with a static analysis that facilitates the inference of resource boundaries [45]. The recent approaches combined session types [47] with automated type-based resource inference [28], making the resource consumption to be a part of an interaction protocol between different contracts and their users.

Gas Analysis in Account/Balance model. The primary goal of estimating the execution cost (in terms of gas) for a transaction involving a smart contract is to predict the amount of digital currency one needs to pay to miners for the transaction processing. However, as the recent study shows, in an *account/balance* blockchain model such costs are typically parametric in the values of the transaction parameters as well as in the values of certain blockchain components, which might not be known at the moment when the transaction is broadcast [83]. As a simple example, imagine a contract, whose execution depends on the value of a block number, in which the transaction is going to be adopted, or on the value of a state component of another contract, which will have changed between the moment the transaction is proposed and the moment it is processed.

Therefore, contrary to the common perception that the main virtue of a sound and complete gas analyser for smart contracts is to predict the *exact* dynamic gas consumption [61, 92], we believe the main benefit of such an analysis is the possibility to detect gas inefficiency patterns prior to contract deployment [22]. For instance, the gas analyser shipped along with Scilla programming language may return the $\top$ result when it fails to infer a polynomial boundary [83]. Assuming the soundness of the analysis, the $\top$ result of our analysis is still informative, as it indicates worse-than-linear gas consumption, which is usually a design flaw.

Gas Analysis in UTxO model. In comparison with the account/balance model, gas analysis in the UTxO model is relatively straightforward, and is much more precise, as it is able to provide the *exact* execution cost. This is due to the fact that in UTxO each transaction, when proposed, should specify its "predecessor" transactions in the adopted history so far, thus, fixing its input values at that moment. The obvious downside of such an execution model is the execution bottleneck it poses: if several transactions that depend on the same output are proposed concurrently, at most one of them will be adopted. At the same time, the non-determinism of the account/balance model allows all concurrently-proposed non-conflicting transactions to be committed within the same block.

1.6.2 GAS CONSUMPTION AND COMPILATION

Gas costs need to be stated in terms of a certain operational model and fixed for each execution unit. In this regard, EVM's design is the most uncontroversial, as it puts gas costs on the most primitive commands. However, this complicates the gas analysis for any of the high-level languages that are compiled to EVM. Not only they are imprecise, and modification and optimisation in a compiler pipeline might break those the analysis logic, that will have to be adapted accordingly.

What if the high-level contract language is what's being deployed on the blockchain? Such a design decision, adopted, for instance, by Scilla, has made it possible to state the gas boundaries in terms of the operational semantics describing the evaluation of contracts in terms of the high-level language [83]. This choice can potentially face a dual threat: the high-level gas boundaries can be invalidated if an efficient optimising compiler is implemented that significantly reduces the projected execution costs of certain commands.

That said, even though EVM is a low-level language, it is prone to the similar potential issue, as it is no longer directly interpreted, but is compiled *just-in-time* as a contract is being executed [36], hence the gas costs defined by its specification might be rendered inadequate. As another recent case study shows, this conjecture is proven in practice, and the current ill-defined gas costs in Ethereum open it to certain kinds of denial-of-service attacks [70].

One of the most promising approaches to balance the discrepancy between the high-level and the low-level gas consumption is to employ the idea of Typed Assembly Language [65] and type-preserving compilation. Wang's PhD Thesis [92] explores this direction and achieves very promising results showing that one can have high-level resource boundaries to be accurate translated to the boundaries for the low-level code. The main disadvantages of the TAL-based resource analysis and compilation are (a) the requirement to have an expressive enough (and usually very hard-to-design) type system for the target language and (b) the need to preserve the typing information across compiler optimisations.

1.7 STANDING RESEARCH PROBLEMS

Before concluding, we provide a list of what we believe are important (and yet unsolved) problems in the programming language design for smart contracts.

Equivalence of account/balance and UTxO models. The choice of language abstractions and static reasoning mechanisms to the large extent is determined by the state model of the underlying consensus protocol: UTxO or account/balance. While the former provides better opportunities for parallel execution and encourages functional programming style, it makes it difficult to implement arbitrary state and, specifically, encode custom currencies in addition to the native one. That said, to date it is unclear whether the choice of the underlying execution model impacts the expressivity of the application layer. In other words, the conjecture of whether the same set of contract-enabled applications can be encoded on top of both UTxO and account/balance model.

Equivalence of bank-centric and account-centric models. Move's approach to managing assets is different from those of Solidity, Scilla, Obsidian and Flint's in the sence that it allocates the resources (*i.e.*, the amounts of custom currencies) with the accounts they belong to and piggy-backs on the runtime's mechanism to ensure the absence of duplication. In contrast, other languages' approach forces each contract that introduces a custom notion of a token to provide the functionality that controls the distribution of the tokens between the users of the contract. Two models, therefore, appear to be complementary: while in account-centric model, the currency definition is provided the rules only for splitting and joining the funds, while the virtual machine takes care of enforcing the linearity, in the contract-centric model all the accounting logic is implemented by the contracts themselves, which play the roles of independent "banks". A theoretical question to be answered is whether these two models are equivalent in terms of expressivity, and also with regard to their desired security properties, and, if so, can one automatically translate digital contracts implemented in one of those paradigms into another one.

Adequate accounting of gas costs. We believe that the gas-aware nature of smart contract programming poses some of the most interesting unsolved challenges in the PL design and implementation.

The first problem that needs to be solved is addressing the discrepancies between the high-level and low-level languages in terms of the projected gas consumption and the definitions of the gas costs. While we mention some ways to achieve that in § 1.6.2, they are by no means a final solution. To wit, they cannot account for the specialised hardware that might be employed by certain miners to decrease the intrinsic cost of transaction processing, thus, giving then some unfair advantage. Therefore, the ideal gas costs need to be *amortised* with regard to multiple ways of executing the same contract commands.

The other challenge is to extend the gas allocation to other aspects of contract-related transaction processing. For instance, EVM only charges gas for dynamic contract execution, while Scilla also mandates the miners to type-check the contracts upon deployment, as well as validate all the messages for the type information. This imposes additional intrinsic transaction processing costs, which do not fit the gas model of the smart contract language, yet need to be accounted for the same reasons the gas price is put on contract executions. In the future, we foresee more validation checks will need to be added to the mining routine. This is why a principled (non-*ad-hoc*) solution for converting *intrinsic* execution costs of mining to the *extrinsic* gas prices is very much desired.

1.8 CONCLUSION

Peter Landin's seminal paper **The Next 700 Programming Languages** [57] stated the following seminal thesis: *"[...] we must systematise [language] design so that a new language is a point chosen from a well-mapped space, rather than a laboriously devised construction"*. This thesis is indeed applicable to the specific family of programming languages used for implementing digital contracts.

We have surveyed what the essential concepts of smart contract intrinsics: *atomicity*, *communication*, *asset management*, and *resource accounting*. We believe that any language for smart contracts coming in the future will have to provide suitable abstractions for expressing and manipulating those concepts, which serve as a set of basic blocks for building reliable and trustworthy blockchain applications.

Bibliography

[1] Securify. https://securify.chainsecurity.com/, accessed on Jan 2, 2019.

[2] æternity Blockchain. Sophia, 2019. https://github.com/aeternity/protocol/blob/master/contracts/sophia.md.

[3] G. A. Agha. *ACTORS - a model of concurrent computation in distributed systems*. MIT Press series in artificial intelligence. MIT Press, 1990.

[4] M. Al-Bassam, A. Sonnino, S. Bano, D. Hrycyszyn, and G. Danezis. Chainspace: A sharded smart contracts platform. In *NDSS*. The Internet Society, 2018.

[5] E. Albert, P. Gordillo, A. Rubio, and I. Sergey. Running on Fumes - Preventing Out-of-Gas Vulnerabilities in Ethereum Smart Contracts Using Static Resource Analysis. In *VECoS*, volume 11847 of *LNCS*, pages 63–78. Springer, 2019.

[6] J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *OOPSLA (Companion)*, pages 1015–1022. ACM, 2009.

[7] G. Alfour. Introducing LIGO: a new smart contract language for Tezos, 2019. https://medium.com/tezos/introducing-ligo-a-new-smart-contract-language-for-tezos-233fa17f21c7.

[8] J. 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/.

[9] L. Alt and C. Reitwießner. SMT-Based Verification of Solidity Smart Contracts. In *ISoLA*, volume 11247 of *LNCS*, pages 376–388. Springer, 2018.

[10] S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *CPP*, pages 66–77. ACM, 2018.

[11] M. D. Angelo and G. Salzer. A Survey of Tools for Analyzing Ethereum Smart Contracts. In *IEEE International Conference on Decentralized Applications and Infrastructures*, *DAPPCON*, pages 69–78. IEEE, 2019.

[12] J. Armstrong. A history of Erlang. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, pages 1–26. ACM, 2007.

[13] N. Atzei, M. Bartoletti, and T. Cimoli. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *POST*, volume 10204 of *LNCS*, pages 164–186. Springer, 2017.

[14] S. Bano, A. Sonnino, M. Al-Bassam, S. Azouvi, P. McCorry, S. Meiklejohn, and G. Danezis. SoK: Consensus in the Age of Blockchains. pages 183–198, 2019.

[15] K. Bansal, E. Koskinen, and O. Tripp. Automatic Generation of Precise and Useful Commutativity Conditions. In *TACAS*, volume 10805 of *LNCS*, pages 115–132. Springer, 2018.

[16] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Zanella-Béguelin. Formal Verification of Smart Contracts: Short Paper. In *PLAS*, pages 91–96. ACM, 2016.

[17] Bitcoin Wiki. Bitcoin Script, 2017. https://en.bitcoin.it/wiki/Script, accessed on Apr 5, 2019.

[18] S. Blackshear, E. Cheng, D. L. Dill, V. Gao, B. Maurer, T. Nowacki, A. Pott, S. Qadeer, Rain, D. Russi, S. Sezer, T. Zakian, and R. Zhou. Move: A language with programmable resources, 2019. https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf.

[19] V. Buterin. A Next Generation Smart Contract & Decentralized Application Platform, 2013. https://www.ethereum.org/pdfs/EthereumWhitePaper.pdf/.

[20] M. M. T. Chakravarty, J. Chapman, K. MacKenzie, O. Melkonian, M. P. Jones, and P. Wadler. The extended UTXO model. volume 12063 of *LNCS*. Springer, 2020.

[21] J. Chang, B. Gao, H. Xiao, J. Sun, and Z. Yang. scompile: Critical path identification and analysis for smart contracts. *CoRR*, abs/1808.00624, 2018.

[22] T. Chen, X. Li, X. Luo, and X. Zhang. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*, pages 442–446. IEEE Computer Society, 2017.

[23] M. Coblenz. Obsidian: A Safer Blockchain Programming Language. In *ICSE (Companion)*, pages 97–99. IEEE Press, 2017.

[24] M. J. Coblenz, R. Oei, T. Etzel, P. Koronkevich, M. Baker, Y. Bloem, B. A. Myers, J. Sunshine, and J. Aldrich. Obsidian: Typestate and assets for safer blockchain programming. *CoRR*, abs/1909.03523, 2019.

[25] Coq Development Team. *The Coq Proof Assistant Reference Manual - Version 8.10*, 2019. http://coq.inria.fr.

[26] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*, pages 238–252. ACM, 1977.

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

[28] A. Das, S. Balzer, J. Hoffmann, and F. Pfenning. Resource-aware session types for digital contracts. *CoRR*, abs/1902.06056, 2019.

[29] A. Das, J. Hoffmann, and F. Pfenning. Work analysis with resource-aware session types. In *LICS*, pages 305–314. ACM, 2018.

[30] M. 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.

[31] C. Dong, Y. Wang, A. Aldweesh, P. McCorry, and A. van Moorsel. Betrayal, Distrust, and Rationality: Smart Counter-Collusion Contracts for Verifiable Cloud Computing. In *CCS*, pages 211–227. ACM, 2017.

[32] Ethereum Foundation. Decentralized Autonomous Organization, 2018. https://www.ethereum.org/dao.

[33] Ethereum Foundation. Vyper, 2018. https://vyper.readthedocs.io.

[34] Ethereum Foundation. Solidity documentation, 2019. http://solidity.readthedocs.io.

[35] E. Foundation. Safety – Ethereum Wiki, 2018. https://github.com/ethereum/wiki/wiki/Safety.

[36] E. Foundation. The Ethereum EVM JIT, 2019. https://github.com/ethereum/evmjit.

[37] J. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[38] N. Grech, M. Kong, A. Jurisevic, L. Brent, B. Scholz, and Y. Smaragdakis. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL*, 2(OOPSLA):116:1–116:27, 2018.

[39] I. Grishchenko, M. Maffei, and C. Schneidewind. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST*, volume 10804 of *LNCS*, pages 243–269. Springer, 2018.

[40] S. Grossman, I. Abraham, G. Golan-Gueta, Y. Michalevsky, N. Rinetzky, M. Sagiv, and Y. Zohar. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL*, 2(POPL), 2018.

[41] E. Gün Sirer. Reentrancy Woes in Smart Contracts, 2016.

[42] P. Haller and F. Sommers. *Actors in Scala - concurrent programming for the multi-core era*. Artima, 2011.

[43] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[44] Y. Hirai. Bamboo, 2018. https://github.com/pirapira/bamboo.

[45] J. Hoffmann, A. Das, and S. Weng. Towards automatic resource bound analysis for OCaml. In *POPL*, pages 359–373. ACM, 2017.

[46] J. Hoffmann and Z. Shao. Type-based amortized resource analysis with integers and arrays. In *FLOPS*, volume 8475 of *LNCS*, pages 152–168. Springer, 2014.

[47] K. Honda, V. T. Vasconcelos, and M. Kubo. Language primitives and type discipline for structured communication-based programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer, 1998.

[48] IOHK Foundation. Marlowe: A Contract Language For The Financial World, 2019. https://testnet.iohkdev.io/marlowe/.

[49] IOHK Foundation. Plutus: A Functional Contract Platform, 2019. https://testnet.iohkdev.io/plutus/.

[50] J. Johannsen and A. Kumar. Introducing the ZIL Cashflow Smart Contract Analyser, 2019. Blog post available at https://blog.zilliqa.com/introducing-the-zil-cashflow-smart-contract-analyser-ded8b4d84362.

[51] S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. In *NDSS*, 2018.

[52] A. Kennedy. Relational parametricity and units of measure. In *POPL*, pages 442–455. ACM Press, 1997.

[53] A. Kiayias, A. Russell, B. David, and R. Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol. In *CRYPTO, Part I*, volume 10401 of *LNCS*, pages 357–388. Springer, 2017.

[54] E. Kokoris-Kogias, P. Jovanovic, L. Gasser, N. Gailly, E. Syta, and B. Ford. Omniledger: A secure, scale-out, decentralized ledger via sharding. In *2018 IEEE Symposium on Security and Privacy, SP*, pages 583–598. IEEE Computer Society, 2018.

[55] A. Kolluri, I. Nikolic, I. Sergey, A. Hobor, and P. Saxena. Exploiting the laws of order in smart contracts. In *ISSTA*, pages 363–373. ACM, 2019.

[56] J. Krupp and C. Rossow. teether: Gnawing at ethereum to automatically exploit smart contracts. In *USENIX Security Symposium*, pages 1317–1333. USENIX Association, 2018.

[57] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[58] L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS*, pages 254–269. ACM, 2016.

[59] L. Luu, V. Narayanan, C. Zheng, K. Baweja, S. Gilbert, and P. Saxena. A secure sharding protocol for open blockchains. In *CCS*, pages 17–30. ACM, 2016.

[60] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. *CWI Quarterly*, 2:219–246, 1989.

[61] M. Marescotti, M. Blicha, A. E. J. Hyvärinen, S. Asadi, and N. Sharygina. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *ISoLA*, volume 11247 of *LNCS*, pages 450–465. Springer, 2018.

[62] P. McCorry, A. Hicks, and S. Meiklejohn. Smart Contracts for Bribing Miners. In *Financial Cryptography and Data Security - FC 2018 International Workshops*, volume 10958 of *LNCS*, pages 3–18. Springer, 2019.

[63] P. McCorry, S. F. Shahandashti, and F. Hao. A smart contract for boardroom voting with maximum voter privacy. In *FC*, volume 10322 of *LNCS*, pages 357–375. Springer, 2017.

[64] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer, 1980.

[65] J. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to Typed Assembly Language. In *POPL*, pages 85–97. ACM, 1998.

[66] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. Abailable at http://bitcoin.org/bitcoin.pdf.

[67] I. Nikolić, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *ACSAC*, pages 653–663. ACM, 2018.

[68] OCaml PRO. Liquidity, 2019. https://www.liquidity-lang.org/.

[69] R. O'Connor. Simplicity: A New Language for Blockchains, 2017. https://blockstream.com/simplicity.pdf.

[70] D. Pérez and B. Livshits. Broken metre: Attacking resource metering in EVM. *CoRR*, abs/1909.07220, 2019.

[71] A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. VerX: Safety Verification of Smart Contracts. In *IEEE Symposium on Security and Privacy SP*, 2020.

[72] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[73] G. Pîrlea and I. Sergey. Mechanising Blockchain Consensus. In *CPP*, pages 78–90. ACM, 2018.

[74] RChain Cooperative. Rholang, 2019. https://rholang.rchain.coop.

[75] C. Reitwiessner. Babbage—a mechanical smart contract language, 2017. Online blog post.

[76] M. Rodler, W. Li, G. O. Karame, and L. Davi. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *NDSS*, 2019.

[77] F. Schrans. Writing safe smart contracts in flint. Master's thesis, Imperial College London, Department of Computing, 2018.

[78] F. Schrans, S. Eisenbach, and S. Drossopoulou. Writing safe smart contracts in Flint. In <*Programming*> *(Companion)*, pages 218–219. ACM, 2018.

[79] P. L. Seijas, S. J. Thompson, and D. McAdams. Scripting smart contracts for distributed ledger technology. *IACR Cryptology ePrint Archive*, 2016.

[80] I. Sergey and A. Hobor. A Concurrent Perspective on Smart Contracts. In *WTSC*, volume 10323 of *LNCS*, pages 478–493. Springer, 2017.

[81] I. Sergey, A. Kumar, and A. Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR*, abs/1801.00687, 2018.

[82] I. Sergey, A. Kumar, and A. Hobor. Temporal Properties of Smart Contracts. In *ISoLA*, volume 11247 of *LNCS*, pages 323–338. Springer, 2018.

[83] I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao. Safer smart contract programming with Scilla. *PACMPL*, 3(OOPSLA):185:1–185:30, 2019.

[84] F. Sun. UTXO vs Account/Balance Model, 2018. Online blog post, available at https://medium.com/@sunflora98/utxo-vs-account-balance-model-5e6470f4e0cf.

[85] N. Szabo. Smart Contracts, 1994. Online manuscript.

[86] Tezos Foundation. Michelson: the language of Smart Contracts in Tezos, 2018. http://tezos.gitlab.io/mainnet/whitedoc/michelson.html.

[87] S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *WETSEB@ICSE*, pages 9–16. ACM, 2018.

[88] A. Trunov. A scilla vs move case study, 2019. Blog post available at https://medium.com/@anton_trunov/a-scilla-vs-move-case-study-afa9b8df5146.

[89] P. Tsankov, A. M. Dan, D. Drachsler-Cohen, A. Gervais, F. Bünzli, and M. T. Vechev. Securify: Practical security analysis of smart contracts. In *CCS*, pages 67–82. ACM, 2018.

[90] D. N. Turner, P. Wadler, and C. Mossin. Once upon a type. In *FPCS*, pages 1–11. ACM, 1995.

[91] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science 1993, 18th International Symposium, MFCS'93*, volume 711 of *LNCS*, pages 185–210. Springer, 1993.

[92] P. Wang. *Type System for Resource Bounds with Type-Preserving Compilation*. PhD thesis, Massachusetts Institute of Technology, 2019.

[93] P. Wang, D. Wang, and A. Chlipala. TiML: a functional language for practical complexity analysis with invariants. *PACMPL*, 1(OOPSLA):79:1–79:26, 2017.

[94] G. Wood. Ethereum: A Secure Decentralized Generalised Transaction Ledger, 2014.

[95] Zilliqa Team. The Zilliqa Technical Whitepaper, 2017. Version 0.1.