Safer Smart Contract Programming with Scilla

ILYA SERGEY, Yale-NUS College, Singapore and National University of Singapore, Singapore
VAIVASWATHA NAGARAJ, Zilliqa Research, India
JACOB JOHANNSEN, Zilliqa Research, Denmark
AMRIT KUMAR, Zilliqa Research, United Kingdom
ANTON TRUNOV, Zilliqa Research, Russia
KEN CHAN GUAN HAO, Zilliqa Research, Malaysia

The rise of programmable open distributed consensus platforms based on the blockchain technology has aroused a lot of interest in replicated stateful computations, *aka smart contracts*. As blockchains are used predominantly in financial applications, smart contracts frequently manage millions of dollars worth of virtual coins. Since smart contracts cannot be updated once deployed, the ability to reason about their correctness becomes a critical task. Yet, the de facto implementation standard, pioneered by the Ethereum platform, dictates smart contracts to be deployed in a low-level language, which renders independent audit and formal verification of deployed code infeasible in practice.

We report an ongoing experiment held with an industrial blockchain vendor on designing, evaluating, and deploying Scilla, a new programming language for safe smart contracts. Scilla is positioned as an intermediate-level language, suitable to serve as a compilation target and also as an independent programming framework. Taking System F as a foundational calculus, Scilla offers strong safety guarantees by means of type soundness. It provides a clean separation between pure computational, state-manipulating, and communication aspects of smart contracts, avoiding many known pitfalls due to execution in a byzantine environment. We describe the motivation, design principles, and semantics of Scilla, and we report on Scilla use cases provided by the developer community. Finally, we present a framework for lightweight verification of Scilla programs, and showcase it with two domain-specific analyses on a suite of real-world use cases.

CCS Concepts: • **Software and its engineering → Functional languages**; **Distributed programming languages**; • **Theory of computation → Program analysis**.

Additional Key Words and Phrases: Blockchain, Smart Contracts, Domain-Specific Languages, Static Analysis

ACM Reference Format:
Ilya Sergey, Vaivaswatha Nagaraj, Jacob Johannsen, Amrit Kumar, Anton Trunov, and Ken Chan Guan Hao. 2019. Safer Smart Contract Programming with Scilla. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 185 (October 2019), 30 pages. https://doi.org/10.1145/3360611

1 INTRODUCTION

Smart contracts are self-enforcing, self-executing protocols governing an interaction between several (mutually distrusting) parties. Initially proposed by Szabo (1994), this idea could only be implemented in a practical setting more than fifteen years later, with the rise of open byzantine consensus protocols powered by the blockchain technology (Bano et al. 2017; Pîrlea and Sergey

Authors' addresses: Ilya Sergey, Yale-NUS College, Singapore, National University of Singapore, Singapore, ilya.sergey@yale-nus.edu.sg; Vaivaswatha Nagaraj, Zilliqa Research, India, vaivaswatha@zilliqa.com; Jacob Johannsen, Zilliqa Research, Denmark, jacob@zilliqa.com; Amrit Kumar, Zilliqa Research, United Kingdom, amrit@zilliqa.com; Anton Trunov, Zilliqa Research, Russia, anton@zilliqa.com; Ken Chan Guan Hao, Zilliqa Research, Malaysia, ken.changuanhao@gmail.com.

2475-1421/2019/10-ART185
https://doi.org/10.1145/3360611

2018). While simple forms of smart contracts were already available for regulating exchange of virtual coins in earlier cryptocurrencies such as Bitcoin (Nakamoto 2008), smart contracts owe their wide adoption to the Ethereum framework (Wood 2014). Ethereum-style smart contracts can be thought of as replicated reactive objects that can store arbitrary state and execute arbitrary computations. Each interaction of an end client with a smart contract happens in a transaction, during which more contracts can be invoked transitively. In addition to performing replicated computations, such transactions result in altering the state and transferring coins between the smart contracts, and also in transferring coins between accounts, which might belong to both end users and contracts—all those changes reproduced across the entire shared blockchain state.

The design of its run-time environment for smart contracts—the Ethereum Virtual Machine (EVM)—adopted a number of choices striving to find a balance between (a) expressivity, (b) performance and (c) safety. To account for (a), EVM offers a Turing-complete low-level language, whose features, amongst others, include 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, and even fraud schemes. The performance aspect (b) has been addressed by designing EVM so it would be a suitable source for just-in-time compilation, thus allowing one to substitute a reference interpreter with an optimised back-end at the client side. Finally, the safety aspect (c) has not received a lot of attention, with the rationale that any failed smart contract execution would simply lead to a transaction that initiates it not taking any effect.

Having a very expressive language with weak safety guarantees has led to the discovery of a number of vulnerabilities and potential exploits in smart contract implementations (Atzei et al. 2017; Luu et al. 2016), with some of them resulting in the loss of tens of millions of US dollars worth of Ethereum currency (Alois 2017; del Castillo 2016). The fact that most of those contracts were written in a high-level language, Solidity, and compiled down to EVM has contributed to this state of affairs, due to multiple bugs discovered in the compiler itself (*cf.* Known Bugs in Solidity Compiler). Finally, since contracts must be deployed as EVM bytecode, independent audit of potentially malicious third-party contracts has proved difficult.

The research community has enthusiastically responded to the challenge of reasoning about Ethereum smart contracts with an explosion of tools and techniques for verification and detection of vulnerabilities, via SMT (Alt and Reitwießner 2018; Bansal et al. 2018; Marescotti et al. 2018) and symbolic execution (Chang et al. 2018; Kalra et al. 2018; Kolluri et al. 2018; Krupp and Rossow 2018; Luu et al. 2016; Marescotti et al. 2018; Nikolić et al. 2018), dynamic (Grossman et al. 2018) and static analysis (Grech et al. 2018; Tikhomirov et al. 2018; Tsankov et al. 2018), and certified programming (Amani et al. 2018; Bhargavan et al. 2016; Grishchenko et al. 2018).

We believe that the area of smart contracts is still in its youth. In this work we have therefore decided to venture in a different direction, giving priority to the safety concern (c), and rethinking Ethereum's takes on (a) and (b) by proposing new foundations for blockchain-based programming.

1.1 Our Proposal

As the main purpose of smart contracts is to provide a transparent way to implement decentralised accounting, with safety being our key concern, we depart from Ethereum's low-level Turing-complete execution model. As an alternative, we propose SCILLA: a novel intermediate-level functional smart contract programming language. By "intermediate" we mean that SCILLA is deliberately minimalistic, and implements exactly its formalisation, reducing syntactic sugar to the necessary minimum in the tradition of intermediate representations adopted by optimisers (Peyton

Jones 1987) and verified compilers (Kumar et al. 2014), and allowing for deployment of executable contract *source* code (as opposed to EVM low-level bytecode) to the blockchain. Scilla aims to achieve both *sufficient expressivity* and *tractability*, while enabling formal contract verification, by adopting the following design principles based on separation of programming concerns:

- **Separating computation from communication.** Scilla contracts are structured as *communicating automata*; every in-contract computation (*e.g.*, changing its balance or computing the result of a function) is implemented as a standalone, atomic *transition*, *i.e.*, without involving any other parties. 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 side-stepping large classes of vulnerabilities (Gün Sirer 2016) and providing a principled reasoning mechanism about safety and temporal properties.
- **Pure and effectful computations.** Any in-contract computation happening within a transition has to terminate, and has to have a predictable effect on the state of the contract and the execution. To this end, we draw inspiration from functional programming with effects, making a distinction between pure expressions (*e.g.*, expressions with primitive data types), impure local state manipulations (*i.e.*, reading/writing into contract state) and blockchain reflection (*e.g.*, reading the current block number).

For Scilla's pure expression fragment, in which most of the computations happen, we adopt a version of the polymorphic lambda calculus (System F) (Girard 1972; Reynolds 1974)—a choice dictated by its foundational properties (*e.g.*, parametric polymorphism), the abundance of interactive theorem provers supporting it, and a large body of research on static analysis and compilation techniques for functional languages having System F in their core (Kumar et al. 2014; Peyton Jones 1987). We guarantee that *every contract transition terminates* by excluding general recursion from the language, and providing instead a set of structural recursion schemes (*aka fold functionals*, or folds (Danvy and Spivey 2007)).

We argue for the viability of our proposal by delivering on the following in application to Scilla:

- ***Safety.*** Scilla provides standard memory and execution safety guarantees by adopting and extending the type theory of System F. It also employs the type checker to ensure the validity of inter-contract interactions.
- ***Minimalism.*** The Scilla reference interpreter and type checker are only a few dozen lines of OCaml code each. The interpreter implements a modular semantics for failure tracking and accounting for resource usage, allowing for inexpensive construction of static analysers.
- ***Expressiveness.*** Despite the lack of general recursion, Scilla has been successfully used to implement all classes of most commonly deployed smart contracts, including auctions, ICOs,[1] wallets, and multiplayer games.
- ***Verification friendliness.*** We have built, on top of Scilla, a framework implementing analyses for custom domain-specific properties, which we showcase by developing and applying two concrete analyses, described in Sec. 5: (a) resource consumption and (b) tracking cash-flows in the state of a contract. We foresee the straightforward embedding of Scilla into existing proof assistants (*e.g.*, Coq), although the development of such an embedding is future work.

1.2 Pragmatic and Conceptual Contributions

The central pragmatic contributions of our work are:

- the design and implementation of Scilla, a minimalistic type-safe functional language for smart contracts, equipped with a formal semantics from the start,

[1] *Initial Coin Offering*, a form of a crowdfunding-implementing contract.

- a report on the experience to date of using Scilla in production and by the community developers, as deployed on top of a realistic blockchain protocol, and
- a discussion on the adopted and rejected choices in the smart contract language design, as well as their implications for the user experience, performance, and the overall blockchain ecosystem.

In this work, we do not claim to introduce any novel programming language mechanisms. Instead, we show that by building on a foundational calculus one can achieve the expressive power necessary for implementing many kinds of realistic applications in a new emerging problem domain.

The minimalistic nature of Scilla and its well-defined semantics enables the following conceptual contributions, advancing the state of the art in smart contract programming:

- a generic and extensible framework for lightweight verification of smart contracts by means of user-defined domain-specific analyses, with inter-stage dependencies, and
- an instantiation of the framework with two domain-specific analyses: (a) worst-case *resource consumption* and (b) tracking of *cash-flows* within a contract, and
- an extensive evaluation of Scilla *wrt.* performance and tractability of the contracts written in it, the latter assessed via a suite of tailored domain-specific analyses (a) and (b).

We argue that our language design facilitates the principled development of the resource analysis (a), while also enabling the cash-flow analysis (b), which, to the best of our knowledge, is novel.

2 OVERVIEW

Scilla is an explicitly-typed functional programming language in the ML family, with higher-order functions, an imperative fragment, and explicit effects, encoding common operations for blockchain applications. Fig. 1 presents the entire abstract syntax of Scilla, outlining its primitive and algebraic types (Fig. 1a), standard constructors for algebraic data types (Fig. 1b), built-in operations (Fig. 1c), pure expressions (Fig. 1d), imperative statements and, finally, contracts (Fig. 1e). As an abbreviation for an ordered vector of similar syntactic atoms of the kind x, we use the notation $\overline{x_j}$ to indicate lifting to tuples, with j ranging over the elements of the vector. For instance, in the case of multiple typed identifiers, $\overline{i_j} : \overline{t_j}$ stands for $i_1 : t_1, \ldots, i_N : t_N$ for some $N \geq 1$. We will be omitting the indexing subscript (*i.e.*, j in this case) if the nature of the multiple entries of a vector is clear from the context. The grayed elements of the syntax correspond to either the artifacts of the reference interpreter implementation (*cf.* Sec. 3), or have to do with the implementation of general recursion, which the smart contract programmer has no access to. Both those cases are explained below.

In the remainder of this section we describe the main aspects of Scilla, with examples in a user-friendly ML-style notation, explaining the most prominent design choices and their implications *wrt.* the run-time behaviour of contracts, as well as their safety guarantees in a series of *Intermezzos*.

2.1 Contracts as State-Transition Systems

Since smart contracts are generally perceived as reactive autonomous agents, Scilla syntactically represents them as state-transition systems, loosely similar to I/O Automata (Lynch and Tuttle 1989). Specifically, every interaction with a contract, potentially resulting in the changes of its mutable state, receiving and transferring the funds to other parties (user accounts or contracts) is implemented by means of *transitions*. Each transition is triggered by a *message* from a user account or another contract, whose delivery is supplied by the underlying blockchain protocol (more on that in Sec. 3.5). To handle the incoming messages, each transition has a unique name within a contract, which messages aiming for it must refer to, as well as a *signature*, describing the types of the expected message components, which can be then used within the transition implementation. Each transition is executed within a transaction atomically, *i.e.*, without transferring control to other contracts, and results in a possible modification of the mutable state of the contract, the transfer

$$\begin{array}{lrcl}
\text{(signed integers)} & int & ::= & \mathsf{i32} \mid \mathsf{i64} \mid \mathsf{i128} \mid \mathsf{i256} \\
\text{(unsigned integers)} & uint & ::= & \mathsf{u32} \mid \mathsf{u64} \mid \mathsf{u128} \mid \mathsf{u256} \\
\text{(byte strings)} & bst & ::= & \mathsf{bystrx}\ n \mid \mathsf{bystr} \\
\text{(primitive types)} & pt & ::= & int \mid uint \mid bst \mid \\
 & & & \mathsf{string} \mid \mathsf{bnum} \mid \mathsf{msg} \\
\text{(algebraic types)} & \mathcal{D} & ::= & \mathsf{unit} \mid \mathsf{bool} \mid \mathsf{nat} \mid \mathsf{option} \mid \\
 & & & \mathsf{pair} \mid \mathsf{list} \mid \mathsf{U} \\
\text{(general Types)} & t & ::= & pt \mid \mathsf{map}\ t\ t \mid t \to t \mid \\
 & & & \mathcal{D}\ \overline{t} \mid \alpha \mid \mathsf{forall}\ \alpha.\ t
\end{array}$$

(a) Types

$$\begin{array}{lrcl}
\text{(unit)} & \mathsf{unit} & ::= & \mathsf{Unit} \\
\text{(booleans)} & \mathsf{bool} & ::= & \mathsf{True} \mid \mathsf{False} \\
\text{(Peano numbers)} & \mathsf{nat} & ::= & \mathsf{Zero} \mid \mathsf{Succ\ nat} \\
\text{(options)} & \mathsf{option}\ \alpha & ::= & \mathsf{None} \mid \mathsf{Some}\ \alpha \\
\text{(pairs)} & \mathsf{pair}\ \alpha_1\ \alpha_2 & ::= & \mathsf{Pair}\ \alpha_1\ \alpha_2 \\
\text{(lists)} & \mathsf{list}\ \alpha & ::= & \mathsf{Nil} \mid \mathsf{Cons}\ \alpha \\
\text{(variants)} & \mathsf{U}\ \overline{\alpha} & ::= & \mathsf{C}_1\ \overline{\alpha} \mid \ldots \mid \mathsf{C}_n\ \overline{\alpha}
\end{array}$$

(b) Algebraic data type definitions

$$\begin{array}{lrcl}
\text{(strings)} & blt_{\mathsf{string}} & ::= & \mathbf{concat} \mid \mathbf{substr} \mid \mathbf{strlen} \mid \mathbf{to\_string} \\
\text{(blocks)} & blt_{\mathsf{bnum}} & ::= & \mathbf{blt} \mid \mathbf{badd} \mid \mathbf{bsub} \\
\text{(hashes)} & blt_{hash} & ::= & \mathbf{sha256} \mid \mathbf{keccak256} \mid \mathbf{ripemd160} \mid \mathbf{to\_bystr} \mid \mathbf{schnorr\_verify} \mid \mathbf{ecdsa\_verify} \\
\text{(maps)} & blt_{\mathsf{map}} & ::= & \mathbf{contains} \mid \mathbf{put} \mid \mathbf{get} \mid \mathbf{remove} \mid \mathbf{to\_list} \mid \mathbf{size} \\
\text{(numeric)} & blt_{num} & ::= & \mathbf{add} \mid \mathbf{sub} \mid \mathbf{mul} \mid \mathbf{div} \mid \mathbf{rem} \mid \mathbf{pow} \\
\text{(integers)} & blt_{int} & ::= & \mathbf{to\_int32} \mid \mathbf{to\_int64} \mid \mathbf{to\_int128} \mid \mathbf{to\_int256} \\
\text{(u. integers)} & blt_{uint} & ::= & \mathbf{to\_uint32} \mid \mathbf{to\_uint64} \mid \mathbf{to\_uint128} \mid \mathbf{to\_uint256} \mid \mathbf{to\_nat} \\
\text{(built-ins)} & blt & ::= & \mathbf{eq} \mid blt_{\mathsf{string}} \mid blt_{\mathsf{bnum}} \mid blt_{hash} \mid blt_{\mathsf{map}} \mid blt_{num} \mid blt_{int} \mid blt_{uint}
\end{array}$$

(c) Built-in operations

$$\begin{array}{lrcl}
\text{(identifiers)} & i, c & ::= & \text{alpha-numeric string} \\
\text{(Values)} & v & ::= & str :: \mathsf{string} \mid \kappa :: int \mid \upsilon :: uint \mid b :: \mathsf{bnum} \mid bsx\ n\ s :: \mathsf{bystrx}\ n \mid bs\ s :: \mathsf{bystr} \mid \\
 & & & ms\ (\overline{str} \mapsto \overline{v}) :: \mathsf{msg} \mid mp\ t_k\ t_\upsilon\ (\overline{v_k} \mapsto \overline{v_\upsilon}) :: \mathsf{map}\ t_k\ t_\upsilon \mid d :: \mathcal{D}\ \overline{t} \mid \\
 & & & clo :: \mathsf{Value} \longrightarrow \mathsf{EvalRes\ Value} \mid tclo :: \mathsf{Type} \longrightarrow \mathsf{EvalRes\ Value} \\
\text{(patterns)} & pat & ::= & \_ \mid i \mid \mathbf{constr}\ c\ \overline{pat} \\
\text{(Expressions)} & e & ::= & \mathbf{val}\ v \mid \mathbf{var}\ i \mid \mathbf{message}\ (\overline{str} \mapsto \overline{i}) \mid \mathbf{constr}\ c\ \overline{t}\ \overline{i} \mid \mathbf{builtin}\ blt\ \overline{i} \mid \\
 & & & \mathbf{let}\ i = e_1\ \mathbf{in}\ e_2 \mid \mathbf{fun}\ (i : t) \Rightarrow e \mid \mathbf{app}\ i\ \overline{i_j} \mid \mathbf{tfun}\ \alpha \Rightarrow e \mid \mathbf{inst}\ i\ \overline{t} \mid \\
 & & & \mathbf{match}\ i\ \overline{pat} \Rightarrow \overline{e} \mid \mathbf{fix}\ (i : t,\ e)
\end{array}$$

(d) Values and expressions

$$\begin{array}{lrcl}
\text{(statements)} & s & ::= & i_1 \leftarrow i_2 \mid i_1 := i_2 \mid i = e \mid \\
 & & & i_1\ \overline{[i_k]}\ := i_2 \mid i_1 \leftarrow i_2\ \overline{[i_k]} \mid i_1 \leftarrow \mathbf{exists}\ i_2\ \overline{[i_k]} \mid \\
 & & & \mathbf{delete}\ i_1\ \overline{[i_k]} \mid i_1 \leftarrow \&i_2 \mid \mathbf{accept} \mid \mathbf{send}\ i \mid \mathbf{event}\ i \mid \\
 & & & \mathbf{match}\ i\ \overline{pat} \Rightarrow \overline{s}
\end{array}
\qquad
\begin{array}{lrcl}
\text{(library functions)} & L & ::= & \mathbf{let}\ i = e \\
\text{(fields)} & F & ::= & i_f : t_f = e_f \\
\text{(transitions)} & T & ::= & \langle i_T, \overline{i_j : t_j}, \overline{s} \rangle \\
\text{(contracts)} & C & ::= & \langle i_C, \overline{L}, \overline{i_p : t_p}, \overline{F}, \overline{T} \rangle
\end{array}$$

(e) Statements, transitions, and contracts

Fig. 1. Abstract syntax of Scilla. Grayed parts are not available at the program level.

and/or acceptance of funds, the emission of a series of new messages *to be sent*, and zero or more *events*, used to inform the external blockchain clients about certain outcomes of the interaction.

Intermezzo 1 (On DAO and Reentrancy Vulnerability). The DAO[2] vulnerability, which is one of the most famous exploits in the history of Ethereum smart contracts, was caused by the fact that a contract can transfer control to another, potentially malicious contract in the midst of its own execution by simply *calling* the other contract as a function. That would allow a malicious contract to call the vulnerable contract *back*, thus potentially exploiting the consequences of the vulnerable

[2] Decentralised Autonomous Organization (Ethereum Foundation 2018a).

```
library Crowdfunding
(*  Map ByStr20 Uint128 → ByStr20 → Uint128 → *)
(*  Option (Map ByStr20 Uint128)              *)
let check_update = (* ... *)
(*  BNum → BNum → Bool  *)
let blk_leq = (* ... *)

contract Crowdfunding
(* Immutable parameters *)
(owner : ByStr20, max_block : BNum, goal : Uint128)
(* Mutable fields *)
field backers : Map ByStr20 Uint128 = Emp ByStr20 Uint128
field funded : Bool = False
(* Transitions *)
transition Donate (sender : ByStr20, amount : Uint128)
transition GetFunds (sender : ByStr20, amount : Uint128)
transition ClaimBack (sender : ByStr20, amount : Uint128)
```

Fig. 2. A signature of the `Crowdfunding` contract.

contract being in an *intermediate* state of the computation. This behaviour, dubbed *reentrancy* (Gün Sirer 2016), in the case of DAO led to the draining of USD 50 million worth of cryptocurrency, and resulted in a fork in the blockchain going against the consensus protocol, as well as sparking a lot of research on ensuring future contract implementations being *reentrancy-safe* (Grossman et al. 2018; Rodler et al. 2019; Tsankov et al. 2018). In contrast, Scilla sidesteps the reentrancy issue by design; by making the message-passing communication the only way for the contracts to interact, we enforce atomicity of changes in a contract state.[3]

For an intuition of a contract layout, consider a `Crowdfunding` smart contract. The goal of the contract is, as the name implies, to collect donations aiming for a certain goal by a specified deadline, given as a "maximal" block number in the underlying blockchain.[4] It should then allow potential backers to donate certain amounts of funds, making records of those donations. If the goal is reached by the deadline, the `owner` of the contract, specified upfront via its account address, should be able to extract the funds, at which point the fulfillment of their obligations to the backers is no longer a concern that could be addressed via the blockchain. If the goal is not reached before the deadline, each backer should be able to claim their donation back.

Fig. 2 shows a high-level signature of `Crowdfunding` with executable code omitted. The prelude of the contract defines the **`library`** of the *pure* (*i.e.*, side effect-free) functions that the contract can use to perform computations on the data stored in its state, in a referentially-transparent way (Mitchell 2003). We elide the implementations of the library functions `check_update` (used to conditionally update the map of backers and their donations) and `blk_leq` (used to compare two block numbers) and only show their types, whose meaning should be clear. Each contract's pure library can be referred by other contracts, independently deployed later, enabling reuse of the code, shared by means of the replicated blockchain state (as will be described Sec. 3.5). The `Crowdfunding` contract, though, does not rely on any external functions.

Intermezzo 2 (External Libraries and Parity Wallet hack). Another famous hack in Ethereum, resulting USD 146 million worth of coins becoming inaccessible (Alois 2017), was caused by

[3] Indeed, other concurrency-related issues, *e.g.*, caused by non-determinism of transaction scheduling, remain to be present even in the "transition-as-an-atomic-change" model adopted by Scilla. However, detecting those issues requires more domain-specific input from the user (Kolluri et al. 2018), and, we believe, should be addressed at a higher-level by means of a suitable domain-specific language for particular smart contract scenarios (*e.g.*, interacting with an off-chain oracle)

[4] This is sound, as block numbers grow monotonically, without gaps.

mutability and *side-effects* in libraries implemented by third-party contracts.[5] Problems of this kind are avoided in SCILLA by making libraries *immutable* and purely functional, while keeping them reusable by other contracts. Such client contracts, when being deployed, have to explicitly *link* to the libraries available on the blockchain.

Following the **library** is the contract signature itself. It starts from the three *immutable* contract parameters: the owner address (represented by a byte string of length 20), the deadline (max_block), and the desired goal. Then follows two fields constituting the contract's mutable state; a map backers containing the record of donations, and the boolean flag funded indicating whether the campaign has succeeded.

Finally, the contract "body" is the three transitions, corresponding to the intended functionality outlined above. Each transition expects exactly two fields to be present in the incoming message, and the implementation treats them as formals of the transition. In the deployed language we allow for sender and amount specifically to be omitted from a transition signature, as they are supplied by the underlying blockchain back-end with the values of the corresponding types.

2.2 Imperative Fragment

The listing on the next page provides a large part of the implementation of the Donate transition of the Crowdfunding contract, which performs a number of necessary checks, and then records the new donation by updating the map of backers. This logic necessitates changes to the contract's fields, and requires information about the outcome of the interaction to flow to the outside world, so the logic falls into the *imperative* fragment of the language.

SCILLA, being an intermediate language, enforces the programs written in A-Normal Form (Flanagan et al. 1993), where each result of a computation, pure or imperative, is used after having been assigned to an immutable variable. The imperative code in Fig. 3 first reads the number of the block, corresponding to the transaction executing the current interaction (& BLOCKNUMBER) into a variable blk. It then assigns the boolean result of invoking the library function blk_leq to in_time, which is subsequently scrutinised in a pattern-matching statement (in SCILLA, pattern matching generalises the conditional statements, as booleans are treated uniformly with other algebraic data types). The first branch, corresponding to still ongoing campaign (True) proceeds with reading the contents of the field backers into bs and conditionally updating it via check_update. Finally, in the case of successful update, the new donation map bs1 is written back to backers, the donation is accepted via the **accept** command, and the event e, indicating the successful donation, is issued.

```
blk ← & BLOCKNUMBER;
in_time = blk_leq blk max_block;
match in_time with
| True  ⇒
  bs ← backers;
  res = check_update bs sender amount;
  match res with
  | Some bs1 ⇒
    backers := bs1;
    accept;
    e = {eventname : "Donated"; donor : sender; ...};
    event e
(* more instructions for other branches *) end end
```

Fig. 3. A fragment of a Donate transition.

Intermezzo 3 (On Accepting Money Explicitly). Initially, in Ethereum any successful call to a smart contract by a client would result with the former "silently" accepting the funds. This is not always a welcome behaviour, and it might lead to the loss of money. As the result, this has lead to introducing the **payable** modifier into SOLIDITY, Ethereum's programming language, along with a number of related conventions.[6] In contrast, SCILLA's program-level **accept** command, provides a finer granularity for controlling such an important event as a transfer of funds.

[5] https://medium.com/@Pr0Ger/another-parity-wallet-hack-explained-847ca46a2e1c

[6] https://solidity.readthedocs.io/en/v0.5.7/contracts.html#fallback-function

Striving for simplicity, Scilla provides only a few mechanisms for structured imperative control flow: straight-line command sequences, and pattern-matching. There are no loops, no effectful recursion, and no calls to procedures that change the contract's state. Finally, for efficiency reasons, Scilla features (amortised) constant-time read and update of `Map`-typed fields of a contract, by referring to the (possibly nested) entries of a map field `f` as `f[k1]...[kn]`, as shown in Fig. 1e.

2.3 Expressions

Most of the computations in Scilla take place in its *pure* fragment, similar in its syntax to the Haskell Core language (Peyton Jones 2013). Expressions serve as contract library functions, field initialisers, and right-hand sides of statements of the form $x = e$ which assign the result of evaluating e to a single-assigned, immutable local variable x.

Primitive types and built-in operations. For the contracts to interact with the back-end blockchain layer, we introduce a number of primitive data types and built-in operations on them. All such data types are listed in Fig. 1a, and the operations on them are provided in Fig. 1c.

The treatment of signed and unsigned integers (*int* and *uint*) of different bit depths is standard, except for the fact that an integer overflow causes a run-time exception, and thus requires explicit checks in a contract implementation. Of some interest are the two types of byte strings, bystrx n and bystr. The former is a family of (value-)dependent types, parameterised by the length of the string and allowing for operations (*e.g.*, for equality via **eq**) only with the values of the same type. The type of addresses in the underlying blockchain is represented by values of type `ByStr20` (Fig. 2). Some byte strings, however, do not have a length that is known statically (*e.g.*, input to hashing via the **keccak256** operation), hence they are given type bystr. Those byte strings can still be passed around as values and compared with other values of bystr. However, length-dependent instances of bystrx n need to be first cast to bystr via **to_bystr** in order to be treated as such.

Some built-in operations from Fig. 1c are polymorphic in the types of their input arguments, and the type of their result is determined by the type of the inputs. Examples of such are **add**, **mul** and other operations that manipulate both signed and unsigned integers alike, but require the arguments to have the same type, which would be the same as the type of the result. Similarly, the overloaded **eq** compares for equality on instances of the same data type. The type system of Scilla, described in Sec. 4, takes this built-in polymorphism into account.

Algebraic data types. Scilla comes with a number of the most common predefined Algebraic Data Types (ADTs), all listed in Fig. 1b, as well as a mechanism to describe user-defined polymorphic variant types, which are non-recursive. Two specific ADTs, nat and list form the basis for encoding iteration in Scilla, which otherwise has no mechanism for looping and general recursion. Specifically, both nat and list come equipped with *structural recursion principles*, both implemented as higher-order combinators (considered as implicitly imported functions from Scilla standard library), whose polymorphic type signatures are as follows:

```
nat_fold   : forall 'A.    ('A → Nat → 'A) → 'A → Nat → 'A
list_foldl: forall 'A 'B. ('A → 'B → 'B) → 'B → (List 'A) → 'B
list_foldr: forall 'A 'B. ('B → 'A → 'B) → 'B → (List 'A) → 'B
```

In case of list, the recursion principles correspond to two ways of "folding" the list, dubbed *fold-left* and *fold-right*, and for nat, just one recursion principle is provided since the two left/right-styles of "folding" a natural number are equivalent (Danvy 2019). Internally, all three recursion schemes are implemented using the fixpoint combinator **fix** $(i : t, e)$ for general recursion (*cf.* Fig. 1d), which is *not available* at the level of client programs and whose semantics is standard (Sec. 3). That said, the recursion principles of Scilla are provably terminating via a syntactic measure, and as such are provided to the end programmers.[7]

Intermezzo 4 (Predictable gas consumption). A contract execution can be interrupted if the amount of the computational resources it has consumed (*aka gas*) exceeds a certain limit (Wood 2014). General recursion and **while**-loops make it difficult to reason about gas consumption, and also makes contracts prone to so-called "out-of-gas"-related exploits (Chen et al. 2017; Grech et al. 2018). By replacing loops and recursion with structural folds, whose consumption depends only on the consumption of the iterated function and the size of a data value, Scilla enables effective static analysis of gas usage (*cf.* Sec. 5.1).

The folds are sufficient to implement a rich standard library of higher-order functions for manipulating lists and natural numbers. To wit, Fig. 4 shows an implementation of the canonical `list_map`, which is explicitly polymorphic, having `'A` and `'B` as its type parameters. All type applications in Scilla are explicit (*i.e.*, there is no implicit elaborations (Pollack 1990) except for built-ins): type variables `'A` and `'B` are used to instantiate the typing schema of `list_foldr` (`@`-syntax stands for the **inst** $i\ \overline{t}$ form from Fig. 1d), as well as the `List` constructors `Cons` and `Nil`.

```
(* forall 'A. forall 'B. ('A → 'B) → List 'A → List 'B *)
let list_map = tfun 'A ⇒ tfun 'B ⇒
  fun (f : 'A → 'B) ⇒ fun (l : List 'A) ⇒
  let folder = @list_foldr 'A (List 'B) in
  let init = Nil {'B} in
  let iter = fun (h : 'A) ⇒ fun (z : List 'B) ⇒
    let h1 = f h in
    Cons {'B} h1 z
  in folder iter init l
```

Fig. 4. An implementation of List's `map` combinator.

3 EXECUTION SEMANTICS

The big-step semantics for Scilla contracts is provided by a reference big-step monadic definitional interpreter (Reynolds 1998), which is currently employed to execute contract-affecting transactions on top of our host blockchain protocol. In our description of Scilla executions, we give up the customary formalism for big-step semantics, which is known to suffer from explosion of the number of rules in the presence of run-time failures and other threaded computational effects (Charguéraud 2013). Instead, we present our big-step semantics with possible run-time failures in a more concise, (but, arguably, less orthodox) monadic Haskell-like style (Owens et al. 2016; Shali and Cook 2011), while explaining, in plain English, the semantics of the involved meta-functions.

3.1 Evaluation of Expressions and Statements

The semantics of expressions (Fig. 5, top) is defined by the meta-function $\mathcal{E}[\![e]\!]\ \rho$, which maps an expression e and a run-time environment ρ to the evaluation result of meta-type EvalRes Value. Run-time closures (value- and type-parameterised) are represented as the *meta-language functions* with the of type EvalRes Value (the symbol → in their ascribed types is thus a type of meta-functions).

For now, let us take EvalRes to be an `Option`-like type Result α with two constructors: Success α and Failure, for successful and failing computations, correspondingly. The *bind* operation, enabling the Haskell-style **do**-notation chains successful computations, while propagating a failure, and **return** x simply constructs an instance of Success x.

In Fig. 5 failures are only produced *explicitly* in the evaluation rule for **constr** $c\ \overline{t}\ \overline{i}$, but the application of meta-semantic functions may produce failures *implicitly*. For instance, lookup $\rho\ i$ returns a value bound by i in ρ and fails if $i \notin \mathrm{dom}(\rho)$. Similarly, tryApply $f\ \overline{args}$ attempts to

[7] Dependently-typed proof assistants, such as Coq, provide a mechanism for automatically deriving recursion principles for inductively-defined data types, along with the proofs of their termination. In future versions of Scilla we will consider the possibility of implementing this mechanism, even though it might complicate the analyses (Sec. 5).

```
$\boxed{\mathcal{E}\llbracket\cdot\rrbracket\ \cdot\ ::\ \mathsf{Expr} \longrightarrow \mathsf{Env} \longrightarrow \mathsf{EvalRes\ Value}}$

$\mathcal{E}\llbracket\mathbf{message}\ (\overline{str \mapsto i})\rrbracket\ \rho \triangleq \mathbf{do}$
  $\overline{plds} \leftarrow$ mapM (lookup $\rho$) $\bar{i}$
  $\mathbf{return}\ ms\ (\overline{str} \mapsto \overline{plds})$

$\mathcal{E}\llbracket\mathbf{constr}\ c\ \bar{t}\ \bar{i}\rrbracket\ \rho \triangleq \mathbf{do}$
  $\langle cons, \mathcal{D}\rangle \leftarrow$ lookupConstr $c$
  $\mathbf{if}$ arity $cons \neq |\bar{t}|$ $\mathbf{then}$ fail
  $\mathbf{else}$ $\overline{args} \leftarrow$ mapM (lookup $\rho$) $\bar{i}$
        $\mathbf{return}\ cons\ \bar{t}\ \overline{args}$

$\mathcal{E}\llbracket\mathbf{builtin}\ blt\ \bar{i}\rrbracket\ \rho \triangleq \mathbf{do}$
  $\overline{args} \leftarrow$ mapM (lookup $\rho$) $\bar{i}$
  $\overline{tps} \leftarrow$ mapM valType $\overline{args}$
  execBuiltIn $blt\ \overline{tps}\ \overline{args}$

$\mathcal{E}\llbracket\mathbf{let}\ i = e_1\ \mathbf{in}\ e_2\rrbracket\ \rho \triangleq \mathbf{do}$
  $v \leftarrow$ eval $e_1\ \rho$
  eval $e_2\ (\rho \cup [i \mapsto v])$

$\mathcal{E}\llbracket\mathbf{fun}\ (i : \cdot) \Rightarrow e\rrbracket\ \rho \triangleq \mathbf{do}$
  $\mathbf{return}\ \lambda\ arg.$ eval $e\ (\rho \cup [i \mapsto arg])$

$\mathcal{E}\llbracket\mathbf{app}\ i\ \overline{i_j}\rrbracket\ \rho \triangleq \mathbf{do}$
  $\overline{args} \leftarrow$ mapM (lookup $\rho$) $\overline{i_j}$
  $f \leftarrow$ lookup $\rho\ i$
  foldLM tryApply $f\ \overline{args}$

$\mathcal{E}\llbracket\mathbf{fix}\ (i : \cdot, e)\rrbracket\ \rho \triangleq \mathbf{do}$
  $\mathbf{letrec}\ fix = (\lambda\ arg.$
          $f \leftarrow$ eval $e\ (\rho \cup [i \mapsto fix])$
          tryApply $f\ arg)$
  $\mathbf{return}\ fix$

$\mathcal{E}\llbracket\mathbf{val}\ v\rrbracket\ \rho \triangleq \mathbf{return}\ \ v$

$\mathcal{E}\llbracket\mathbf{var}\ i\rrbracket\ \rho \triangleq$ lookup $\rho\ i$

$\mathcal{E}\llbracket\mathbf{tfun}\ \alpha \Rightarrow e\rrbracket\ \rho \triangleq \mathbf{do}$
  $\mathbf{let}\ f = \lambda\ t.$ eval $[t/\alpha]e\ \rho$
  $\mathbf{return}\ f$

$\mathcal{E}\llbracket\mathbf{inst}\ i\ \bar{t}\rrbracket\ \rho \triangleq \mathbf{do}$
  $f \leftarrow$ lookup $\rho\ i$
  foldLM tryInstType $f\ \bar{t}$

$\mathcal{E}\llbracket\mathbf{match}\ i\ \overline{pat \Rightarrow e}\rrbracket\ \rho \triangleq \mathbf{do}$
  $w \leftarrow$ lookup $\rho\ i$
  $(e, [\bar{i} \mapsto \overline{v}]) \leftarrow$ matchPattern $(\overline{pat \Rightarrow e})\ w$
  eval $e\ (\rho \cup [\bar{i} \mapsto \overline{v}])$

$\boxed{\mathcal{S}\llbracket\cdot\rrbracket\ \cdot\ ::\ \mathsf{Stmt} \longrightarrow \mathsf{Conf} \longrightarrow \mathsf{EvalRes\ Conf}}$

$\mathcal{S}\llbracket i_1 \leftarrow i_2\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $v \leftarrow$ load $\sigma\ i_2$
  $\mathbf{let}\ \rho' = \rho \cup [i_1 \mapsto v]$
  $\mathbf{return}\ \langle\rho', \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle$

$\mathcal{S}\llbracket i_1 := i_2\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $v \leftarrow$ lookup $\rho\ i_2$
  $\mathbf{let}\ \sigma' =$ put $\sigma\ i_1\ v$
  $\mathbf{return}\ \langle\rho, \sigma', \beta, \gamma, \iota, \mu, \varepsilon\rangle$

$\mathcal{S}\llbracket i = e\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $v \leftarrow$ eval $e\ \rho$
  $\mathbf{let}\ \rho' = \rho \cup [i \mapsto v]$
  $\mathbf{return}\ \langle\rho', \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle$

$\mathcal{S}\llbracket\mathbf{accept}\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq$
  $\mathbf{return}\ \langle\rho, \sigma, \beta + \iota, \gamma, 0, \mu, \varepsilon\rangle$

$\mathcal{S}\llbracket i_1 \leftarrow \&i_2\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $\upsilon \leftarrow$ lookup $\gamma\ i_2$
  $\mathbf{return}\ \langle\rho \cup [i_1 \mapsto \upsilon], \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle$

$\mathcal{S}\llbracket\mathbf{send}\ i\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq$
  $\upsilon_m \leftarrow$ lookup $\rho\ i$
  $\mathbf{return}\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu\ +\!\!+\ \upsilon_m, \varepsilon\rangle$

$\mathcal{S}\llbracket\mathbf{event}\ i\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq$
  $\upsilon_e \leftarrow$ lookup $\rho\ i$
  $\mathbf{return}\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\ +\!\!+\ \upsilon_e\rangle$

$\mathcal{S}\llbracket\mathbf{match}\ i\ \overline{pat \Rightarrow s}\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $w \leftarrow$ lookup $\rho\ i$
  $(s, [\bar{i} \mapsto \overline{\upsilon}]) \leftarrow$ matchPattern $(\overline{pat \Rightarrow s})\ w$
  seval $s\ \langle\rho \cup [\bar{i} \mapsto \overline{\upsilon}], \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle)$

$\mathcal{S}\llbracket i_1\ \overline{[i_k]}\ :=\ i_2\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $\overline{v_k} \leftarrow$ mapM (lookup $\rho$) $\overline{i_k}$
  $v_\upsilon \leftarrow$ lookup $\rho\ i_2$
  $\sigma' \leftarrow$ updateAsMap $\sigma\ i_1\ \overline{v_k}\ v_\upsilon$
  $\mathbf{return}\ \langle\rho, \sigma', \beta, \gamma, \iota, \mu, \varepsilon\rangle$

$\mathcal{S}\llbracket i_1 \leftarrow i_2\ \overline{[i_k]}\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $\overline{v_k} \leftarrow$ mapM (lookup $\rho$) $\overline{i_k}$
  $v_r \leftarrow$ getAsMap $\sigma\ i_2\ \overline{v_k}$
  $\mathbf{return}\ \langle\rho \cup [i_1 \mapsto v_r], \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle$

$\mathcal{S}\llbracket i_1 \leftarrow \mathbf{exists}\ i_2\ \overline{[i_k]}\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $\overline{v_k} \leftarrow$ mapM (lookup $\rho$) $\overline{i_k}$
  $v_r \leftarrow$ hasAsMap $\sigma\ i_2\ \overline{v_k}$
  $\mathbf{return}\ \langle\rho \cup [i_1 \mapsto v_r], \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle$

$\mathcal{S}\llbracket\mathbf{delete}\ i\ \overline{[i_k]}\rrbracket\ \langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle \triangleq \mathbf{do}$
  $\overline{v_k} \leftarrow$ mapM (lookup $\rho$) $\overline{i_k}$
  $\sigma' \leftarrow$ removeFromMap $\sigma\ i\ \overline{v_k}$
  $\mathbf{return}\ \langle\rho, \sigma', \beta, \gamma, \iota, \mu, \varepsilon\rangle$
```

Fig. 5. Big-step monadic semantics of Scilla expressions (top) and statements (bottom).

apply the closure f (as a meta-function) to a sequence of arguments and fails if the arity of f is not sufficient. valType provides a type of a run-time value. The meaning (and failure model) of the meta-functions lookupConstr and matchPattern should be clear from the context. mapM and foldLM are the standard monadic combinators that lift the failure-threading computations to lists. The meta-function execBuiltIn provides the semantics to built-in operations (with possible arity and run-time type failures) and eval "virtualises" the recursive call to the evaluator itself. For the time being one can think of eval $e\ \rho$ as of an alias for $\mathcal{E}\llbracket e\rrbracket\ \rho$, however, this meaning is going to change with the change of the underlying failure-tracking monad, which we generalise in Sec. 3.3.

The bottom of Fig. 5 shows the evaluation of Scilla statements: $\mathcal{S}\llbracket s\rrbracket$ takes a *configuration* $\langle\rho, \sigma, \beta, \gamma, \iota, \mu, \varepsilon\rangle$ and transforms it into a new one. Besides the environment ρ, a configuration is comprised of the following components: σ is a storage, mapping names of mutable fields to their values; β is a contract's current balance; γ is an environment with immutable (for the duration of the execution) blockchain data; ι is a non-negative number, indicating the "incoming" funds in a message, which the contract might accept; μ is a buffer of outgoing messages to be sent at the end of the transition; ε is a list of events to be emitted. The semantics is mostly self-explanatory,

```
$\mathsf{initLibDef}\ (\mathbf{let}\ i_l = e_l)\ \rho \triangleq \mathbf{do}$
  $v_l \leftarrow \mathsf{eval}\ \rho\ e_l$
  $\mathbf{return}\ (\rho \cup [i_l \mapsto v_l])$

$\mathsf{initField}\ \rho\ (i_f : t_f = e_f)\ \sigma \triangleq \mathbf{do}$
  $v_f \leftarrow \mathsf{eval}\ \rho\ e_f$
  $\mathbf{return}\ \sigma \cup [i_f \mapsto v_f]$

$\mathsf{initContract}\ \langle i_C, \overline{L}, \overline{i_p} : \overline{t_p}, \overline{F}, \overline{T} \rangle\ [\overline{i_q} \mapsto \overline{v_q}]\ \beta \triangleq \mathbf{do}$
  $\rho_{\mathrm{lib}} \leftarrow \mathsf{foldLM}\ \mathsf{initLibDef}\ \emptyset\ \overline{L}$
  $\overline{t_q} \leftarrow \mathsf{mapM}\ \mathsf{valType}\ \overline{v_q}$
  $\mathsf{assert}\ [\overline{i_q} \mapsto \overline{t_q}] = [\overline{i_p} \mapsto \overline{t_p}]$
  $\mathbf{let}\ \rho = \rho_{\mathrm{lib}} \cup [\overline{i_q} \mapsto \overline{v_q}]$
  $\sigma_{\mathrm{init}} \leftarrow \mathsf{foldLM}\ (\mathsf{initField}\ \rho)\ \emptyset\ \overline{F}$
  $\mathbf{return}\ \langle [\overline{i_q} \mapsto \overline{v_q}], \sigma_{\mathrm{init}}, \beta \rangle$
```

```
$\mathsf{checkMsg}\ (\overline{i_j} : \overline{t_j})\ [\overline{i_m} \mapsto \overline{v_m}] \triangleq \mathbf{do}$
  $\overline{t_m} \leftarrow \mathsf{mapM}\ \mathsf{valType}\ \overline{v_m}$
  $\mathsf{assert}\ [\overline{i_j} \mapsto \overline{t_j}] = [\overline{i_m} \mapsto \overline{t_m}]$

$\mathsf{handleMsg}\ \langle i_C, \overline{L}, \overline{i_p} : \overline{t_p}, \overline{F}, \overline{T} \rangle\ \langle \rho_C, \sigma_C, \beta_C \rangle\ \gamma\ \rho_m \triangleq \mathbf{do}$
  $\iota \leftarrow \mathsf{lookup}\ \rho_m\ \text{“}\mathit{amount}\text{”}$
  $i_T \leftarrow \mathsf{lookup}\ \rho_m\ \text{“}\mathit{tag}\text{”}$
  $\langle i_T, \overline{i_j} : \overline{t_j}, \overline{s} \rangle \leftarrow \mathsf{lookup}\ \overline{T}\ i_T$
  $\mathsf{checkMsg}\ (\overline{i_j} : \overline{t_j})\ \rho_m$
  $\langle \cdot, \sigma'_C, \beta_C, \cdot, \iota', \mu', \varepsilon' \rangle \leftarrow$
        $\mathsf{foldLM}\ \mathsf{seval}\ \langle \rho_C \cup \rho_m, \sigma_C, \beta_C, \gamma, \iota, [], [] \rangle\ \overline{s}$
  $\mathbf{let}\ v_{\mathrm{out}} = \sum_{m' \in \mu'} (m'(\text{“}\mathit{value}\text{”}))$
  $\mathsf{assert}\ v_{\mathrm{out}} \leq \beta_C$
  $\mathbf{return}\ \langle \rho_C, \sigma'_C, \beta_C - v_{\mathrm{out}}, \gamma, \iota', \mu', \varepsilon' \rangle$
```

Fig. 6. Contract initialisation (left) and message handling (right).

with a few new meta-functions used for in-place manipulation with a map stored in a field (*i.e.*, updateAsMap, getAsMap, *etc*), and seval, serving as an alias for $\mathcal{S}[\![\cdot]\!]$.

3.2 The Life Cycle of a Contract

When a contract is initialised and deployed to the blockchain (Fig. 6, left), all its libraries (including external ones, *cf.* Sec. 3.5) and fields are bound to the corresponding values.[8] Initialisers for fields can depend on library variables and on the contract's parameters i_p, but not on the values of other fields. The main procedure initContract is invoked by the protocol participants validating the contract-deploying transaction (*aka* miners), with a provided vector of contract argument values $[\overline{i_q} \mapsto \overline{v_q}]$ and balance β. The initialiser checks dynamically (via assert) that the types of provided arguments match the contract signature and aborts the entire transaction if they don't.

Every time a message is received by an initialised contract instance, it is checked and processed via the function handleMsg. Amongst other things, it performs the dynamic check, ensuring that the intended transition (referred to via a message's field "*tag*") indeed exists and that its signature matches the message's components (via checkMsg). It then forms a configuration delegating the execution to seval. Finally, it collects all outgoing transfers of funds from the messages about to be sent into v_{out}, ensuring that the contract has sufficient funds ($v_{\mathrm{out}} \leq \beta_C$), creating the final configuration, handed back to the blockchain layer for further replication.

3.3 Tracking Gas Consumption

So far we have omitted any discussion on an important aspect of blockchain-based replicated computations: resource consumption. A deployment of a contract or interacting with it by sending a message typically requires an emitter to *pay* (in virtual funds) a certain amount of *gas*—resource consumed by the execution. If an execution exceeds the amount of gas allotted by the user, an *out-of-gas* failure is raised. This way, paying for gas when proposing a transaction does not allow the emitter to waste the computational power of other miners by requiring them to perform worthless intensive work. Furthermore, gas fees disincentivise users to consume too much of replicated storage, which is a valuable resource in a replicated state. In this section, we describe Scilla's take on tracking gas consumption.

Monadic gas accounting. The monadic representation of the big-step semantics from Fig. 5 comes with the possibility of adding computational effects to the run-time *modularly* and without altering the core interpreter logic (Liang et al. 1995; Sergey et al. 2013). We take full advantage of this

[8] A pure library can be deployed without an accompanying contract.

Tab. 1. Aspects of gas accounting and their costs.

Static reduction cost, $C_s[\![\cdot]\!] : \text{Expr} \cup \text{Stmt} \to \text{Gas}$	
let $i = e_1$ **in** e_2	1
match $i\ \overline{pat} \Rightarrow \overline{e}$	$\max(\lvert\overline{pat}\rvert) \times \lvert\overline{pat} \Rightarrow \overline{e}\rvert$
fix $(i : t,\ e)$	1
builtin $blt\ \overline{i}$	0
Dynamic built-in cost, $C_b[\![\cdot]\!] : \text{BuiltIn} \to \text{Val}^* \to \text{Gas}$	
concat $v_1\ v_2$	length v_1 + length v_2
sha256v	$\lceil$length (to_string v) $\div$ 64$\rceil \times 15$
keccak256 v	$\lceil$length (to_string v) $\div$ 136$\rceil \times 15$
mul $v_1\ v_2$	$5 *$ max(size_of v_1, size_of v_2)
Dynamic statement cost via monadic operations from Fig. 5	
load $\sigma\ i$	size_of $\sigma(i)$
put $\sigma\ i\ v$	$\lvert$size_of v − size_of $\sigma(i)\rvert$
lookup $\rho\ i$	1
updateAsMap $\sigma\ i\ \overline{v_k}\ \overline{v_v}$	$\overline{v_k} \in$ dom$(\sigma(i))$? size_of $\overline{v_v}$: $\lvert\overline{v_k}\rvert$

possibility by changing the definition of EvalRes Value to mimic the combination of both *failure* and *state* monad, and capture resource consumption, in addition to propagating failure. Specifically, we define EvalRes α as a type synonym for Gas $\longrightarrow$ Result α, that is, making the computations having it as a type to be functions, expecting a certain amount of gas (isomorphic to a natural number), so they can be only *run* when a certain amount g of gas is provided.

This way, each of the monadic operations, as well as a virtualised call to eval and seval from Fig. 5, will subtract from the amount of gas provided for contract initialisation or message handling, so an *out-of-gas* error will be emitted (and propagated further) once the gas supply runs out.

Sources of gas consumption. The gas accounting discipline is implemented by three orthogonal execution components, with examples given in Tab. 1. The first aspect of gas consumption (Tab. 1, top) is due to the cost of reductions of expressions and statements, which is constant for all terms except for pattern matching. Capturing it requires virtualisation of the evaluator calls by means of eval/seval, so the gas-aware versions thereof subtract the corresponding *statically-determined* amount, before passing control to the recursive call of the evaluator.

The second aspect has to do with the cost of executing built-in operations and is depicted by the middle part of Tab. 1. The gas costs in this case are based on the measured relative performance of the operations, as implemented in the reference interpreter. While the costs assigned for most operations are immediate derivations of the sizes of their input arguments, the hash-related built-ins require some care. As their implementations divide their input into blocks (of size 64 and 136 bytes for **sha256** and **keccak256** respectively), the performance of this hashing depends on the number of such blocks, resulting in the corresponding counts.

The bottom of Tab. 1 shows the gas accounting via monadic meta-functions from Fig. 5. Of interest are the operations that modify the storage component σ of the configuration. As those operations must penalise the execution for manipulating with large chunks of the state, the cost for most of them is proportional to the size of the loaded/stored value. That said, lookup has a constant cost, and put only charges for the delta in the old/new sizes of the stored value.

3.4 A Continuation-Passing Style Evaluator

The result EvalRes of evaluating a Scilla program is a composition of a state monad and a failure monad. As the state of the contract grows large (*e.g.*, by storing lists), this choice may cause call stack overflows in the host language, *i.e.*, OCaml.

The state monad introduces *laziness* into the evaluation, wherein each intermediate expression is wrapped into a *thunk*, *i.e.*, a closure whose result depends on the amount of gas provided. Evaluating `let y = f h z in g y` in this monad therefore does not reduce the expression, but instead allocates a closure which contains a reference to the closure that is the result of evaluating `f h z`. When iterated over a list (*e.g.*, via `list_foldl`), this construction thus results in a composition of nested non-tail calls. When finally run, the call stack thus grows in proportion to the length of the list, potentially causing a call stack overflow.

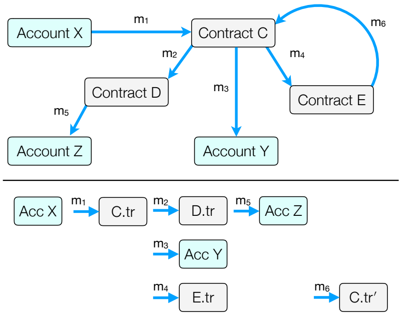
Changing the underlying monad EvalRes to a continuation monad of type $(a \longrightarrow c) \longrightarrow c$ eliminates this problem. The fact that the core evaluator for SCILLA is parametric in an underlying monad implementation means that this change does not involve changes in the core interpreter (Fig. 1), and also allows the result type c to be specialised to embed the logic of both the result and the state monads seamlessly (Filinski 1994). Additionally, since SCILLA is shallowly embedded into OCaml (*i.e.*, SCILLA's run-time closures are OCaml closures), we can exploit the fact that CPS-transforming an interpreter in the host language leads to the interpreted call-by-value language being evaluated in CPS (Danvy and Filinski 1990). This means that the tail-call optimization of the OCaml compiler benefits SCILLA programs, even those that are not written in CPS, thus eliminating the risk of stack overflows. We elaborate on the performance of the CPS evaluator for SCILLA in Sec. 6.2.

3.5 Interaction with the Protocol Layer

The SCILLA evaluator interacts with the underlying blockchain via a JSON interface that abstracts the inner workings of the blockchain layer. The evaluator takes a set of JSON files as input, and on success returns an output JSON to be used by the underlying blockchain layer to update the contract state. An end user interacts with the smart contract layer via transactions. A transaction either deploys a new contract on the blockchain, or invokes a transition on an existing contract. The consensus of the blockchain layer establishes whether the transaction is successful.

Contract deployment. Prior to any execution, the end user must first deploy the contract on the network by submitting a transaction containing (1) the SCILLA contract code, (2) a name-to-address mapping for external libraries used by the contract, (3) actual values for the contract's immutable parameters, and (4) gas to pay the cost of deployment. Once the transaction reaches the network, nodes run a consensus protocol to validate it, during which the submitted contract is checked for type safety (*cf.* Sec. 4). This involves obtaining the cached type signatures for the external libraries the contract depends upon (retrieved via their addresses from the blockchain state), and adding them to the type-checking environment. A check is also performed on the validity of the immutable parameter values. If all the validation checks pass, and sufficient gas is supplied, then the contract and its initialisation parameters get stored at each node in the network. A deployed contract is given an address that is computed using the sender's address. The address uniquely identifies a contract and its library, for those who might want to use them.

Contract execution. Once a contract is deployed on the network, an end user can invoke the contract's transitions by issuing a transaction that specifies (1) the address of the contract, (2) the name of the transition to be invoked, (3) actual parameters to be passed to the transition, (4) the amount of funds to be transferred to the contract, and (5) gas to be paid.



A transition invocation may trigger a chain of contract calls as shown in the figure on the right (top). In case of a multi-contract transaction (*i.e.*, when a contract interacts

Expression typing $\boxed{\Gamma \vdash e : t}$

$$\Gamma \vdash \textbf{val}\ v : \mathsf{valType}\ v \qquad \frac{\forall j, \Gamma \vdash i_j : t_j \quad \mathsf{builtInType}_{\overline{t_j}}\ blt = \overline{t_j} \to t_{\mathrm{res}}}{\Gamma \vdash \textbf{builtin}\ blt\ \overline{i_j} : t_{\mathrm{res}}} \qquad \frac{\forall j, \Gamma \vdash i_j : t_j \quad \Gamma \vdash i : \overline{t_j} \to t_{\mathrm{res}}}{\Gamma \vdash \textbf{app}\ i\ \overline{i_j} : t_{\mathrm{res}}} \qquad \frac{\forall i_j, \Gamma \vdash i_j : t_j \quad \mathsf{storableType}\ t_j}{\Gamma \vdash \textbf{message}\ (\overline{str_j \mapsto i_j}) : \mathsf{msg}}$$

Statement sequence typing $\boxed{\Gamma_e;\Gamma_\sigma;\Gamma_\gamma \vdash \overline{s}}$

$$\frac{\Gamma_\gamma \vdash i_2 : t \quad \Gamma_e, i_1 : t; \Gamma_\sigma;\Gamma_\gamma \vdash \overline{s_j}}{\Gamma_e;\Gamma_\sigma;\Gamma_\gamma \vdash i_1 \leftarrow \&i_2; \overline{s_j}} \qquad \frac{\forall k, \Gamma_e \vdash i_k : t_k \quad \Gamma_\sigma \vdash i_2 : \mathsf{map}\ \overline{t_k}\ t_v \quad \Gamma_e, i_1 : \mathsf{option}\ t_v; \Gamma_\sigma;\Gamma_\gamma \vdash \overline{s_j}}{\Gamma_e;\Gamma_\sigma;\Gamma_\gamma \vdash i_1 \leftarrow i_2\ \overline{[i_k]}; \overline{s_j}} \qquad \frac{\Gamma_e \vdash e : t \quad \Gamma_e, i : t; \Gamma_\sigma;\Gamma_\gamma \vdash \overline{s_j}}{\Gamma_e;\Gamma_\sigma;\Gamma_\gamma \vdash i = e; \overline{s_j}}$$

Transition and contract typing $\boxed{\Gamma_e;\Gamma_\sigma;\Gamma_\gamma \vdash T}\ \boxed{\Gamma_\gamma \vdash C}$

$$\frac{\forall j, \mathsf{storableType}\ t_j \quad \Gamma_e, \overline{i_j : t_j}; \Gamma_\sigma;\Gamma_\gamma \vdash \overline{s}}{\Gamma_e;\Gamma_\sigma;\Gamma_\gamma \vdash \langle i_T, \overline{i_j : t_j}, \overline{s}\rangle} \qquad \frac{\emptyset \vdash \overline{L} \rightsquigarrow \Gamma_{\mathrm{lib}} \quad \forall p, \mathsf{storableType}\ t_p \quad \Gamma_e = \Gamma_{\mathrm{lib}}, \overline{i_p : t_p} \quad \forall f, \Gamma_e \vdash e_f : t_f \quad \forall f, \mathsf{storableType}\ t_f \quad \Gamma_\sigma = \overline{i_f : t_f} \quad \forall j, \Gamma_e;\Gamma_\sigma;\Gamma_\gamma \vdash T_j}{\Gamma_\gamma \vdash \langle i_c, \overline{L}, \overline{i_p : t_p}, \overline{i_f : t_f = e_f}, \overline{T_j}\rangle}$$

Fig. 7. Selected typing rules.

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.[9] 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.

Even though the programming component of Scilla does not include effectful recursion (or loops), in the presence of storable code (contracts), such a recursion can be implemented by means of sending messages by a contract to itself, or via circular communication with other contracts. As this kind of interaction in general cannot be detected locally without disallowing many common scenarios, we have chosen to control it with the fixed size of a transaction's message chain. If the chain length limit is reached, the whole transaction rolls back.

4 TYPE SYSTEM AND BASIC CONTRACT VALIDATION

Contracts available on blockchain come from different sources, and both contract developers and clients should be able to guarantee basic safety properties of the code to be executed in a transaction. Scilla comes with a simple type system, mostly inherited from System F, allowing blockchain users to *validate* a contract before it is deployed and used. Selected typing rules are given in Fig. 7 with Γ ranging over standard typing contexts. Values are typed via the meta-function valType, familiar from Fig. 6; $\mathsf{builtInType}_{\overline{t_j}}\ blt$ takes not only the built-in, but also the vector $\overline{t_j}$ of the inferred argument types, in order to perform the elaboration in the presence of built-in polymorphism. Statements are typed in sequences (Fig. 7, middle), and the transitions are validated with the signatures of the library functions (Γ_{lib}), contract parameters and fields in the typing context (Fig. 7, bottom), altogether constituting to checking contract well-formedness $\Gamma_\gamma \vdash C$.

Both messages and contract state must be possible to serialise, in order to store them on a blockchain. Combining first-class functions and "storability" in a type-safe way is known to be difficult (Leifer et al. 2003), as these two features put together make it possible to, *e.g.*, capture a part of context information within a closure and send it in a message. To circumvent this issue, Scilla's type system uses the predicate storableType, whose definition (elided for brevity) outlaws any types having a function or a type abstraction as its component. It is used in Fig. 7 for validating contracts and messages. This check is sufficient to guarantee *statically* that no non-"storable" value (*e.g.*, a function) would flow to a message or a field. The type system ensures the absence of run-time errors in well-typed contracts, except for the following failure classes:

[9] Breadth-first was chosen over depth-first because it provides better fairness guarantees for sequential message processing.

Definition 4.1 (Expected failure). We say that a Scilla expression execution eval $e\ \rho\ g$ or a statement sequence execution (foldLM seval $\langle \rho, \sigma, \beta, \gamma, \iota, [], [] \rangle\ \overline{s}\ g$), for the corresponding input environment/configuration and an amount g of *gas*, results in an *expected failure* is it fails due to either (i) non-exhaustive pattern matching, (ii) Out-of-Gas failure, or (iii) integer overflow failure.

Our type soundness argument is worth mentioning. As the operational semantics of Scilla (Fig. 5) is in big-step style, it comes with the explicit notion of a failure (via EvalRes monad), allowing one to distinguish between run-time errors and non-termination and formulate the type safety result (Harper 2012, Section 7.3) (*i.e.*, well-typed contracts don't go wrong, but can fail with an expected failure). But how would one establish this result for a *big-step* semantics (as the "progress-and-preservation" approach (Wright and Felleisen 1994) does not apply in this case)?

The problem of conducting such a proof has been explored before (Amin and Rompf 2017; Owens et al. 2016), and the solution is to instrument the big-step semantics with *"execution fuel"*, which is then used as an inductive argument for proving type soundness (Siek 2012). Luckily for us, Scilla semantics, when made gas-aware (Sec. 3.3), gets the desired fuel instrumentation, so it suffices to prove that for a well-typed expression e and any amount of gas g, the (non-CPS) evaluation of $\mathcal{E}[\![v]\!]\ \rho\ g$ can only fully reduce to Success v, or result in an expected failure. The proof of this statement follows the approach, described by Siek (2013), in the same way as it has been used for establishing type soundness with definitional (big-step) interpreters in prior work (Amin and Rompf 2017). The type soundness *wrt.* the CPS evaluator (Sec. 3.4) follows from the fact that CPS-transformation is semantics-preserving (Danvy and Filinski 1992).

We conclude this section with the two important corollaries of Scilla type soundness, reflecting on the guarantees the type system provides *wrt.* contract initialisation and handling messages. Both of them use an auxiliary definition toTypeEnv ρ, which returns a typing context Γ by applying valType component-wise to values in all entries of ρ.

Proposition 4.2 (Initialisation type soundness). *If* $\Gamma_\gamma \vdash \langle i_c, \overline{L}, \overline{i_p} : \overline{t_p}, \overline{i_f} : \overline{t_f} = \overline{e_f}, \overline{T_j} \rangle$, *and* $[\overline{i_q} \mapsto \overline{v_q}]$ *is such that* toTypeEnv $[\overline{i_q} \mapsto \overline{v_q}] = \overline{i_p} : \overline{t_p}$, *then for any* β *and* g, *the execution of* initContract $\langle i_C, \overline{L}, \overline{i_p} : \overline{t_p}, \overline{F}, \overline{T} \rangle\ [\overline{i_q} \mapsto \overline{v_q}]\ \beta\ g$,

- *either results in* $\langle \rho, \sigma_{\mathrm{init}}, \beta \rangle$, *such that* toTypeEnv $\rho = \overline{i_p} : \overline{t_p}$ *and* toTypeEnv $\sigma_{\mathrm{init}} = \overline{i_f} : \overline{t_f}$, *or*
- *terminates with an expected failure.*

Proposition 4.3 (Type soundness for message handling). *If* $\Gamma_\gamma \vdash \langle i_C, \overline{L}, \overline{i_p} : \overline{t_p}, \overline{i_f} : \overline{t_f} = \overline{e_f}, \overline{T_j} \rangle$, *and* $\langle \rho_C, \sigma_C, \beta_C \rangle\ \gamma$ *are such that* toTypeEnv $\rho_C = \overline{i_p} : \overline{t_p}$ *and* toTypeEnv $\sigma_C = \overline{i_f} : \overline{t_f}$, *then for any values* β *and* g, *the execution of* handleMsg $\langle i_C, \overline{L}, \overline{i_p} : \overline{t_p}, \overline{F}, \overline{T} \rangle\ \langle \rho_C, \sigma_C, \beta_C \rangle\ \gamma\ \rho_m\ g$

- *either results in* $\langle \rho_C, \sigma'_C, \beta'_C, \gamma, \iota', \mu', \varepsilon' \rangle$, *such that* toTypeEnv $\sigma'_C = \overline{i_f} : \overline{t_f}$, $\beta'_C \geq 0$,
- *halts due to exception raised at any line of* handleMsg, *except for the one executing* seval, *or*
- *terminates with an expected failure.*

5 SUPPORT FOR LIGHTWEIGHT VERIFICATION

The type safety results enabled by Propositions 4.2 and 4.3 rule out many classes of run-time bugs, yet leave a possibility for a contract to fail with, *e.g.*, non-exhaustive pattern matching or running out of gas. To provide even stronger static safety guarantees, without elaborating the type system (as complex types might harm the adoption), but by means of lightweight (*i.e.*, fully automated) verification, we have built an extensible framework of *pluggable staged checkers* (Dietl et al. 2011), allowing third-party users to develop their own static analyses. As an instance of the framework, we developed a pattern-matching exhaustiveness checker, following the standard algorithm for ML (Sestoft 1996). Other simple checkers were contributed by community members.

$$\frac{\Phi \vdash e_1 \rightsquigarrow \langle \overline{i_{e_1}}; s_1; g_1 \rangle \qquad \Phi, i : \langle \overline{i_{e_1}}; s_1; g_1 \rangle \vdash e_2 \rightsquigarrow \langle \overline{i_{e_2}}; s_2; g_2 \rangle \\ g \triangleq \text{if } \overline{i_{e_1}} = [] \text{ then } g_1 + g_2 \text{ else } 1 + g_2}{\Phi \vdash \textbf{let } i = e_1 \textbf{ in } e_2 \rightsquigarrow \langle []; s_2; g \rangle} \qquad \frac{\Phi \vdash \overline{e_i} \rightsquigarrow \overline{\langle \overline{i_{e_i}}; s_i; g_i \rangle}}{\Phi \vdash \textbf{match } i\ \overline{pat} \Rightarrow \overline{e_i} \rightsquigarrow \mathsf{maxAdd}(\overline{\langle \overline{i_{e_i}}; s_i; g_i \rangle})}$$

$$\frac{\Phi, i : \mathsf{atom} \vdash e \rightsquigarrow \langle \overline{i_j}; s; g \rangle}{\Phi \vdash \textbf{fun } (i : t) \Rightarrow e \rightsquigarrow \langle i, \overline{i_j}; s; g \rangle} \qquad \frac{\Phi \vdash i \rightsquigarrow \phi \qquad \forall j, \Phi \vdash i_j \rightsquigarrow \phi_j \qquad \phi' = \mathsf{applySig}(i, \overline{i_j}, \phi, \phi_j)}{\Phi \vdash i\ \overline{i_j} \rightsquigarrow \phi'}$$

$$\mathsf{applySig}(i, \overline{i_j}, \phi, \overline{\phi_j}) = \begin{cases} \langle [], \mathsf{sapp}\ i\ \overline{i_j}, \mathsf{gapp}\ i\ \overline{i_j} \rangle & \text{if } \phi = \mathsf{atom} \\ \langle [], [\overline{(\text{size } \phi_j)}/\overline{i_j}]s, [\overline{(\text{gas } \phi_j)}/\overline{i_j}]g \rangle & \text{if } \phi = \langle \overline{i_j}; s; g \rangle \end{cases}$$

Fig. 9. Selected rules of the resource analysis: **let**-binding, branching, function definition, and application.

In the remainder of the section, we briefly describe two static analyses for Scilla that we have developed for automated reasoning about *domain-specific* contract properties.

5.1 Compositional Gas Usage Analysis

The first domain-specific instance of the checker framework is the resource analyser, which computes gas usage of a transition as a polynomial of the size of its parameters and contract fields.

```
contract HelloWorld ()
  field welcome_msg : String = ""
  transition SetHello (h : String) (*...*)
  transition SayHello ()
    r ← welcome_msg;
    e = {_eventname: "Hello"; msg: r};
    event e
  end
```

Consider the contract HelloWorld on the right. The cost of executing SayHello can be summarised as the polynomial $a + b$, where a is the size of the string welcome_msg (taken as cost of loading it from the contract state) and b is the cost of creating an event with two strings: "Hello" of constant length and statically unknown welcome_msg. For a statement sequence, the total gas cost is a sum over their individual gas use polynomials (GUP). In the absence of loops and general recursion, worst-case resource usage analysis in Scilla becomes tractable and principled, and does not require state-of-the art techniques, employed, *e.g.*, in resource analysis for OCaml (Hoffmann et al. 2017). Intuitively, a linear (or higher-degree polynomial) cost may only arise from using folds or loading/storing a collection in a contract state.

The gas usage of a higher-order function (*e.g.*, list_map from Fig. 4) may depend on the gas usage of its functional argument, as well as on the *size* of the result value it returns. We capture this by means of *size/gas* signatures ϕ, ascribed to expressions, inspired by the demand analysis of Glasgow Haskell Compiler, GHC (Sergey et al. 2014). A signature ϕ of an expression (Fig. 8) is either atom (indicating an identifier that binds a statically unknown value), or a triple consisting of a parameter vector (for functions), a *size abstraction sr*, and a gas use polynomial *poly*(*gr*).

$$\begin{array}{lrcl} \text{(base values)} & br & ::= & \mathsf{var} \\ \text{(size abstraction)} & sr & ::= & \mathsf{base}\ br \mid \mathsf{len}\ sr \mid \mathsf{elm}\ sr \mid \\ & & & \mathsf{sapp}\ i\ \overline{sr} \mid poly(sr) \mid \\ & & & \mathsf{foldacc}\ i\ \ \overline{sr} \mid \top \mid \ \ldots \\ \text{(gas use polynomial)} & gr & ::= & \mathsf{size}\ sr \mid \mathsf{gapp}\ i\ \overline{gr} \mid poly(gr) \\ \text{(signature)} & \phi & ::= & \mathsf{atom} \mid \langle \overline{i}; sr; poly(gr) \rangle \\ \text{(sign. environment)} & \Phi & ::= & \emptyset \mid \Phi, i : \phi \end{array}$$

Fig. 8. Resource analysis domain

Size abstractions provide a way to express the size of an expression's result in terms of sizes of values, bound by its free variables and their datatype-specific components. For example, elm(X) refers to the elements of a container, *i.e.*, a list or a map. Similarly, len(X) refers to the length of a map or a list. In the presence of higher-orderness, gas usage of an expression may depend on the size of a function application result (in addition to its gas usage), which is not known at the moment of the analysis (think, *e.g.*, a result of applying a parameter function f within list_map). To account for this, size abstraction and GUPs come with terms to express the size/complexity of an application, similar to GHC's *call demands*: sapp and gapp, correspondingly. The first argument of both is a function to be called, the second is an argument vector, with elements of the corresponding kind.

Analysing folds. The reason why resource analysis in Scilla is not entirely trivial is the presence of folds (Fig. 4), which are the modicum of bounded iteration. To tackle them soundly, the resource analyser domain features a special size abstraction for application of folds—foldacc (Fig. 8), capturing the resulting size of the accumulator of a fold operation (over natural numbers or lists). Specifically, in foldacc $i\ \overline{sr}$, i is a reference to the function to be applied iteratively (the "foldee"), and $\overline{sr}$ are the parameters of the fold parameters (including the initial value of the accumulator). When fully applied and processed, this size abstraction expresses the final size of the accumulator of a fold operation over a linear structure, such as a natural number or a list, in terms of the structure's size (*e.g.*, list's length) and the initial value of the accumulator. As common in algorithms with iteration, this requires finding a closed-form solution for a recurrence, arising from the way the accumulator is threaded at each "step" of a fold. At the moment, we only solve constant and first-order linear recurrences (*i.e.*, of the form $f(n+1) = f(n) + \mathit{const}$) to determine the size of the accumulator in a closed-form solution. For other derived recurrences, the analyser returns the top element $\top$, indicating the failure to analyse the gas consumption.

Analysis rules. The analysis for expressions is phrased as an inference system for a judgement $\Phi \vdash e \rightsquigarrow \phi$, which reads as "in the analysis environment Φ, e has the signature ϕ".[10] Some representative rules of the analysis are shown in Fig. 9. For instance, the resource consumption of a function $\textbf{fun}\ (i : t) \Rightarrow e$ is represented by a signature $\langle i, \overline{i_j}; s; g\rangle$, derived from analysing its body and parameterised by i. This signature is "unleashed" when the function is applied to an argument, using the auxiliary function applySig. For the primitive commands and statements, the analysis uses cost assignments of the evaluator (Tab. 1). In the analysis rule for pattern matching, the function maxAdd takes a list of signatures and takes a sum, separately, of the size abstractions and gas use polynomials with the exception that if a polynomial term is present in both the operands of the sum, the maximum of the coefficients is taken, rather than adding them up.

The derived signature for `list_map` (Fig. 4), with the size component omitted, is as follows:

```
Parameter list:  [f, l]
Gas consumption: 5(a) + 1(a)(b) + 11
Legend: a: Length of: l; b: Cost of calling f on (Element of: l)
```

On soundness, completeness, and the virtues of the analyser. The resource analysis is compositional and, hence, is linear in the size of the contract and external libraries it uses (so far we do not cache the analysis results, but this is not difficult to implement). Thanks to the design of Scilla, in which state-manipulating inter-contract calls are impossible by design, the resource analysis of a contract can be done entirely *in isolation*, which is known to be not the case for Ethereum (Wang 2019).

Even though we did not conduct a formal soundness proof, we conjecture that our analyser is sound (*i.e.*, it derives a correct upper boundary on gas consumption), as (a) it employs the costs of primitive operations directly from their gas signatures, (b) it does not under-approximate the results of iteration, and (c) it treats the branching conservatively *wrt.* resource consumption using maxAdd. The analyser is, however, incomplete and does not derive the tightest possible resource bound. The main source of incompleteness is the analyser's inability to solve non-linear recurrences, in which case it returns $\top$. As our experience demonstrates (Sec. 6.1) non-trivial nested loops in contracts are uncommon, and in the current state of the implementation, we are able to analyse all list functions currently being used in contracts developed in-house and the community, except for the library `list_sort` function, which is non-linear in nature. In the future, we are planning to rely on specialised tools for solving recurrences for this purpose (Albert et al. 2008).

[10] We omit the analysis description for statements, which is straightforward.

Contrary to the common perception that the main virtue of a sound and complete gas analyser for smart contracts is to predict dynamic gas consumption (Marescotti et al. 2018; Wang 2019), we believe the main benefit of such an analysis is the possibility to detect gas inefficiency patterns prior to contract deployment (Chen et al. 2017). With this regard, the $\top$ result of our analysis is still informative, as it indicates worse-than-linear gas consumption, which is usually a design flaw.

5.2 Cash-Flow Analysis

The second major application of the checker framework is the cash-flow analysis. Each deployed contract constitutes an independent account on the network, and the contract's transitions can access the current balance of its account through the implicitly declared `balance` field. However, a contract such as the `Crowdfunding` needs to keep track not only of its total balance, but also how much money each of the backers has contributed to the crowdfunding campaign.

The cash-flow analysis 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 we apply standard techniques of abstract interpretation (Cousot and Cousot 1977), 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.

$$\begin{array}{lcl} \tau & ::= & \mathbf{Money} \mid \mathbf{NotMoney} \mid \mathbf{Map}\ \tau \mid t\ \overline{\tau} \mid \top \mid \bot \\ t & ::= & \mathbf{Option} \mid \mathbf{Pair} \mid \mathbf{List} \mid \ \ldots \end{array}$$

$$\begin{array}{lrcll} \text{(maps)} & \mathbf{Map}\ \tau & \sqsubseteq & \mathbf{Map}\ \tau' & \text{iff } \tau \sqsubseteq \tau' \\ \text{(algebraic types)} & t\ \overline{\tau} & \sqsubseteq & t'\ \overline{\tau'} & \text{iff } t = t' \text{ and } \tau_i \sqsubseteq \tau'_i \text{ for all } i \\ \text{(bottom)} & \bot & \sqsubseteq & \tau & \text{for all } \tau \\ \text{(top)} & \tau & \sqsubseteq & \top & \text{for all } \tau \end{array}$$

Fig. 10. Tags and partial ordering on them.

Lattice of tags. The tags (ranged over by τ) mimic the type system and are summarised in Fig. 10. **Money** indicates that an expression represents money; **NotMoney** indicates that an expression *certainly* is not money; **Map** τ is for maps whose co-domain has tag τ;[11] $t\ \overline{\tau}$ indicates that an expression is of the algebraic type t, where the type parameters of t are tagged with $\overline{\tau}$.[12] The meanings of $\bot$ is nothing is known about the component, and $\top$ represents an apparent inconsistency.

The collection of all contract parameters, fields, transition parameters and local parameters, along with their respective tags, form the elements of a lattice with the ordering described in Fig. 10 applied pointwise to each typed AST node. The lattice is finite, since the depth of a combination of **Map** and t is finite for well-typed contracts. The complexity of cash-flow analyser's procedure is determined by the height of the abstract domain lattice and, hence, is quadratic in the size of the program at worst.

Transfer function. The main transfer function (Muchnick 1997) $\rightsquigarrow$ is defined on the lattice of pairs $\Psi, \overline{s}$, where an environment Ψ maps fields and variables to their current tags and $\overline{s}$ is a sequence of statements that are being annotated with tags (Fig. 11). It analyses the usage of variables, and generating new tags representing the least upper bound (*lub*, $\sqcup$) of their current tags and their usage. This constitutes a monotone function within the lattice, and repeated applications of the function (starting from the element where all variables are tagged with $\bot$) are thus guaranteed to reach a fixpoint.

An environment Ψ is threaded through a list of statements, which is analysed *backwards* in order to analyse usage before declarations (Fig. 11, top left). Once a variable declaration i_1 is reached (Fig. 11, top middle), it is tagged with the current tag $\langle i_1, \tau \rangle$, and the variable is removed from the

[11] We are unaware of a use case where a map domain represents money.

[12] We make exceptions for the types nat and bool (as well as user-defined types isomorphic to bool), which in Scilla are algebraic types, but which are treated as base types in the cashflow analysis. Values of those types are tagged with **Money** or **NotMoney** depending on the usage.

$$\frac{\Psi,\ s \rightsquigarrow \Psi'',\ s' \qquad \Psi'',\ \overline{s_j} \rightsquigarrow \Psi',\ \overline{s'_j}}{\Psi,\ \overline{s_j};\ s \rightsquigarrow \Psi',\ \overline{s'_j};\ s'} \qquad \frac{\Psi(i_1) = \tau'_1 \quad \Psi(i_2) = \tau'_2 \quad \tau = \tau'_1 \sqcup \tau'_2 \quad \Psi'' = \mathsf{remove}(i_1, \Psi) \quad \Psi' = \mathsf{replace}(i_2, \tau, \Psi'')}{\Psi,\ \langle i_1, \tau_1 \rangle \leftarrow i_2 \rightsquigarrow \Psi',\ \langle i_1, \tau \rangle \leftarrow i_2} \qquad \frac{\Psi(i) = \tau'' \quad \Psi'' = \mathsf{remove}(i, \Psi) \quad \Psi'' \vdash e \downarrow \tau'' \rightsquigarrow \Psi',\ \langle e', \tau' \rangle}{\Psi,\ \langle i, \tau \rangle = e \rightsquigarrow \Psi',\ \langle i, \tau' \rangle = e'}$$

$$\frac{\Psi(i) = \tau \quad \tau' = \tau_e \sqcup \tau \quad \Psi' = \mathsf{replace}(i, \tau', \Psi)}{\Psi \vdash \mathbf{var}\ i \downarrow \tau_e \rightsquigarrow \Psi',\ \langle \mathbf{var}\ i, \tau' \rangle} \qquad \frac{\psi = \tau_e; \Psi(\bar{i}) \quad \overline{\psi'} = \psi \sqcup \mathsf{sigs}(blt) \quad \psi'' = \bigsqcap \overline{\psi'} \quad \tau'_e; \overline{\tau'} = \psi'' \quad \Psi' = \mathsf{replace}(\bar{i}, \overline{\tau'}, \Psi)}{\Psi \vdash \mathbf{builtin}\ blt\ \bar{i} \downarrow \tau_e \rightsquigarrow \Psi',\ \langle \mathbf{builtin}\ blt\ \bar{i}, \tau'_e \rangle}$$

$$\frac{\begin{array}{c} str_a = \text{“}amount\text{”} \quad \Psi(i_a) = \tau'_a \quad \tau_a = \mathbf{Money} \sqcup \tau'_a \quad \Psi_a = \mathsf{replace}(i_a, \tau_a, \Psi) \\ str_r = \text{“}recipient\text{”} \quad \Psi_a(i_r) = \tau'_r \quad \tau_r = \mathbf{NotMoney} \sqcup \tau'_r \quad \Psi' = \mathsf{replace}(i_r, \tau_r, \Psi_a) \quad \tau'_e = \tau_e \sqcup \mathbf{NotMoney} \end{array}}{\Psi \vdash \mathbf{message}\ (\overline{str} \mapsto \bar{i}) \downarrow \tau_e \rightsquigarrow \Psi',\ \langle \mathbf{message}\ (\overline{str} \mapsto \bar{i}), \tau'_e \rangle}$$

Fig. 11. Selected rules for the cash-flow transfer function.

environment Ψ'. The per-contract field environment is initialised with the implicit field `balance`, mapped to **Money**, the implicit field `this_address`, mapped to **NotMoney**, and with the contract parameters and fields, mapped to $\bot$. The local environment for a transition is initialised with the message fields `amount` and `sender`, mapped to **Money** and **NotMoney** respectively.

In our backwards analysis, expressions are analysed top-down ($\Psi \vdash e \downarrow \tau \rightsquigarrow \Psi', \langle e, \tau' \rangle$) with the use of an *expected* tag τ, which represents a lower tag bound that the expression must have. This bound is derived from the context in which the expression is used in a statement (*cf.* Fig. 11, top-right). The initial sources of the **Money** tag are the `balance` field, and the `amount` fields of incoming and outgoing messages. Whenever these fields are read from or assigned to, the expressions or variables used as the *target* of the read or the source of the assignment are known to represent money. The initial sources of the **NotMoney** tag are the current block number of the blockchain (accessed using `BLOCKNUMBER`), the `this_address` field, and the `sender`/`recipient` message fields.

When variables are used, their usage and their current tags are analysed to determine if their tags need to be changed. For instance, if the variable v with tag **Money** is used as the right-hand side of a map update statement $m\ \overline{[k]}\ := \ v$, then m must have the tag which is greater than or equal to **Map Money**. Similarly, if the variable r with tag **Option NotMoney** is used as the left-hand side of a lookup statement $r \leftarrow m\ \overline{[k]}$, then m must have a tag that is not smaller than **Map NotMoney**.

Arithmetic built-in functions such as `add` and `mul` have multiple consistent usages. For `add` we require both of the arguments and the result to have the same tag, since the addition of **Money** and **NotMoney** is inconsistent. Conversely, applying `mul` to **Money** and **NotMoney** (in any order) is consistent and produces **Money**, whereas applying `mul` to **Money** and **Money** is considered inconsistent. We analyse calls to builtin functions by generating a signature ψ (an expected return tag followed by the list of argument tags) based on the current tags at the call site (Fig. 11, middle-right). We then take the pointwise least upper bound of that signature and all possible signatures of the function being called. The greatest lower bound of the resulting set of signatures constitutes the new tags for the variables used at the call site. If an inconsistency is found in the usage, the variables in question are tagged with **Top** (the *lub* of inconsistent tags).

Example. Running the analysis on the crowdfunding contract (Fig. 2) results in the fields of the contract being tagged as listed in the table on the right. Notice that the `goal` field is being tagged with **Money**. 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 with **Money**, since its value is regularly compared to the value of the `balance` field.

Field/Param	Tag
`owner`	**NotMoney**
`max_block`	**NotMoney**
`goal`	**Money**
`backers`	**Map Money**
`funded`	**NotMoney**

Library functions. Since library functions are pure, each library function defines a relationship between the cashflow tags of actual parameters and the cashflow tags of the result of the function. For built-in functions this relationship is represented using signatures, but establishing a set of consistent signatures for a user-defined library function is non-trivial in general. We therefore conservatively limit the cashflow analysis to the statement part of Scilla, and the simple expressions that are allowed to occur as part of statements.

When applied to contracts used in practice, we observe that this limitation appears to have little adverse impact on the quality of the analysis. The reason for this is that the cashflow relationship between parameters and results of a library function is often trivial in practice, and can thus be determined using only the information from how the parameters and values are used outside of the library function, without a deep analysis of the function body.

Non-native tokens. As mentioned earlier, the initial source of **Money** tags are the `balance` contract field and the `amount` fields of incoming/outgoing messages. However, a common use case for smart contracts is to facilitate the exchange of *non-native tokens*, *i.e.*, tokens that can be used for payment, and thus carry value on their own, but which differ from the virtual currency used by the network that the contract is deployed onto. These non-native tokens are not discovered by the analysis as they are unrelated to and thus never mix with the native tokens in the `balance` and `amount` fields.

The cashflow analysis may be given hints by the user as to which additional fields are used to represent non-native tokens. If a hint is supplied to the analysis, the appropriate field is initially marked as **Money** (or **Map Money**, as appropriate), after which the analysis proceeds as usual.

Future applications. We believe it is reasonable to assume that a contract's money fields must be used consistently as money, and not be mixed with non-money values, in the spirit of type systems with units of measure (Kennedy 1997). To that end, our cashflow analysis is useful to the contract programmer, in that it is able to flag any inconsistent use of money fields.[13] Additionally, we envision an extension to Scilla where a money field can be declared to have a specific relationship with the current balance of the contract.

6 IMPLEMENTATION, EVALUATION, AND ADOPTION

The entire Scilla infrastructure[14] to date is implemented in about 10 kLOC of OCaml with Jane Street extensions to support monadic **do**-notation (Minsky 2016), and 1 kLOC of C++ used for cryptographic primitives. The interpreter interacts with the underlying blockchain layer (implemented entirely in C++) through message passing. Miners that use the standard client application are mandated to run the type-checker (Sec. 4) upon contract deployment, and are encouraged (but not required) to run the analyses described in Sec. 5. The resource and cash-flow analyses are also intended to be used by the contract developer to validate their code with the intent to avoid common mistakes before deployment.

6.1 Our Experience

To evaluate the viability of the language proposal before deploying it to the protocol, we implemented the standard library for manipulating with basic data types (700 LOC of Scilla), and a number of the most commonly used contracts, ported from Ethereum to Scilla.

The statistics for some basic contracts is collected in Tab. 2. It includes widespread applications such as ERC20 and ERC721 (Shirley 2018) tokens, both following the corresponding community

[13] In fact, we have already discovered bugs this way. When the crowdfunding contract was rewritten to use events to signal errors rather than through messages, a money-carrying message was incorrectly changed to be an event. The cashflow analysis suddenly tagged the `backers` field with **Map** $\perp$ rather than **Map Money**, causing the bug to be discovered.

[14] Scilla is available open-source on GitHub: https://github.com/Zilliqa/scilla.

Tab. 2. Statistics for implemented basic contracts

Contract	LOC	#Lib	#Trans	Asympt. GU	$-Flow
HelloWorld	31	3	2	$O(\|\text{string}\|)$	✓
Crowdfunding	127	13	3	$O(\|\text{map}\|)$	✓
Auction	140	11	3	$O(\|\text{map}\|)$	✓
ERC20	158	2	6	$O(1)$	✓*
ERC721	270	15	6	$O(\|\text{map}\|)$	✓$_\perp$
Wallet	363	28	9	$O(\|\text{map}\| \times \|\text{list}\|)$	✓
Bookstore	123	6	3	$O(\|\text{string}\| + \|\text{map}\|)$	✓
HashGame	209	16	3	$O(1)$	✓
Schnorr	71	2	3	$O(\|\text{bystr}\|)$	✓

standards, as well as a version of an auction (Auction), multi-party wallet (Wallet), multi-player game (HashGame), and a service contract for digital signing (Schnorr).[15] The table reports on the contract sizes (in LOC), numbers of library functions each of them defines (#Lib), number of transitions (#Trans), and worst-case asymptotic transition complexity by the gas usage analyser (Sec. 5.1), stated in terms of types ascribed to fields whose size makes the *principal* contribution to gas usage. The last column tells whether the results of the cash-flow analyser (Sec. 5.2) matched the developer's intuition *wrt.* assigning money tags. For all contracts, except for ERC20 and ERC721, the analysis correctly derived the dichotomy (✓) for money/not money-storing fields.

In the two mentioned contracts, fields that store amounts of non-native tokens were marked with $\perp$, as they have no data flow involving `balance`/`amount` components of the contract and messages. That is, the analysis had no "seed" to start assigning the money tags. In the case of the ERC20 contract, once the cashflow analyser was provided with suitable hints regarding which fields should be assigned money tags, the analysis proceeded to mark other fields appropriately as well, thus giving the expected outcome. In the case of the ERC721 contract, however, no extra information was gained even with hints. The reason for this is that ERC721 defines a *non-fungible* type of token, *i.e.*, a type of token where individual tokens are not interchangeable, and which may therefore carry different values. In contrast, the cashflow analyser assumes that all money is *fungible.*[16]

6.2 Performance Evaluation and Comparison to EVM

We evaluated Scilla implementation with the goal of answering the following research questions:

(1) What is Scilla's performance on common contracts and what are the main bottlenecks?
(2) How does the performance of Scilla evaluator compare to EVM on similar contracts?
(3) What are the sizes of similar contracts implemented in Scilla, Solidity, and EVM bytecode?

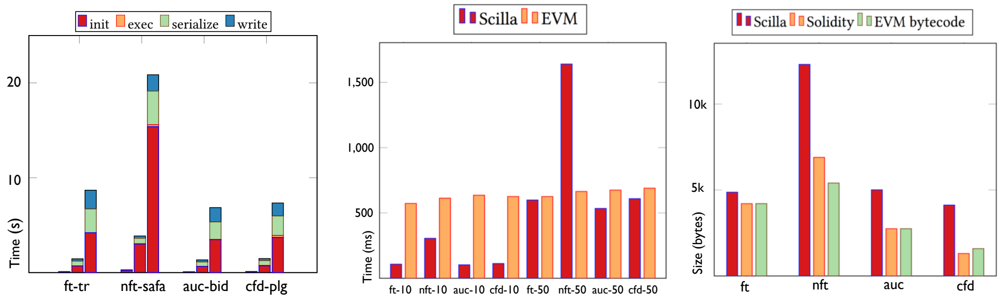
Since we prioritised tractability of the language definition and the contracts implemented in it over performance, it was not our goal to beat EVM, but merely to show that the performance/contract sizes are comparable with those of the highly optimised state-of-the-art blockchain platform. Furthermore, since the main performance bottlenecks in blockchain protocols are the speed of mining transaction and communication costs (taking up to a few seconds), one could afford contract running times that are no worse than the time for creating a new block.

[15] According to https://etherscan.io/tokens, roughly 16% of active smart contracts on Ethereum are ERC20. According to https://dappradar.com/charts, games, gambling, and collectibles (ERC721) are the most used contracts on Ethereum.

[16] A fungible token type is similar to traditional currencies where coins are indistinguishable and thus are freely interchangeable (if they have the same denomination). Conversely, a non-fungible token type is more similar to collector's items, *e.g.*, paintings, where the tokens are distinguishable, and may therefore not be interchangeable.

Tab. 3. Breakdown of contract run-times (in ms): initialisation, execution, serialisation, and output.

	`init`			`exec`			`serialise`			`write`		
Transition/State size	10k	100k	500k	10k	100k	500k	10k	100k	500k	10k	100k	500k
`ft-transfer`	67	709	4,208	0.05	0.05	0.07	25	501	2,506	14	244	1,976
`nft-setApproveForAll`	239	3,011	15,382	1.92	39	206	41	568	3,546	23	242	1,719
`auc-bid`	61	665	3,480	0.83	0.96	0.10	23	456	1,860	17	221	1,515
`cfd-pledge`	68	723	3,705	1.96	42	207	23	500	2,057	20	216	1,368



(a) Relative time breakdown

(b) Scilla/EVM execution times

(c) Code size comparison

Fig. 12. Runtime and size statistics on some representative smart contracts.

For our evaluation we have chosen the most common kinds of contracts used on Ethereum: ERC20 (`ft`), ERC721 (`nft`), auction (`auc`) and crowdfunding (`cfd`). Performance experiments were conducted on a commodity Intel Core i5 machine with 8GB RAM.[17]

To answer question (1), we have evaluated the interpreter performance on *the most expensive* transitions of the chosen contracts (*e.g.*, ERC20's `transfer`), with the size of the largest affected contract state component (*e.g.*, a `map` field) ranging from 10k to 500k entries.[18] The results are shown in Tab. 3 and Fig. 12a. It is clear that the evaluator's performance overhead is negligible (less than 1%) compared to the time taken by input/output of the contract state: reading from blockchain (`init`), serialising and writing it back—those machineries operate with JSON representation of state and their performance deteriorates linearly with the state size. This issue is orthogonal to our study of the language design presented in this paper, and in Sec. 7, we discuss possible ways to address it in the future. That said, even with the suboptimal IO implementation, in most of the cases the observed transaction times are under 10s, which is acceptable for blockchain computations.

The implementation of Scilla is agnostic with regard to the underlying blockchain protocol, and at the moment all interaction is done by passing state snapshots in JSON. Thus, making an apples-to-apples comparison of Scilla/EVM performance is difficult, as EVM is an integral part of the Ethereum protocol, and can access the entire blockchain state in a RAM-like manner. This leads to more slow start-up time for EVM, but nearly constant-time access for contracts with large state, whereas Scilla input-output overhead grows linearly. Fig. 12b shows a comparison of run-times (from the cold start) of Scilla and EVM on the same four contracts with 10k and 50k state entries (first/second four groups). In most of the cases, Scilla's performance is better, but EVM shows superior results, due to more efficient IO, when the state grows beyond 50k entries. The state of `nft` is larger than the projected 10/50k, as it uses nested maps, while we only count "top-level" entries.

[17] The artefact containing the benchmarks is available on GitHub: https://github.com/ilyasergey/scilla-benchmarks.

[18] The largest Ethereum contract to date is ERC20 with 600k entries. Most of deployed contracts have less than 50k entries.

Contract sizes are important, as the miners need to store them locally, increasing the memory overhead of the protocol. Fig. 12c addresses question (3), showing the size (in bytes) of Scilla contracts, which, at worst, is only less than 2.5 times bigger than that of Solidity (which is much more expressive) and of EVM (which is compressed). As the number of deployed contract libraries on the blockchain grows, we expect the average size of a contract to reduce due to code reuse.

6.3 Scilla in the Wild

Scilla has been incorporated into Zilliqa blockchain (Zilliqa Team 2017). Even though Scilla has been released less than a year prior to the submission of the final version of this paper, its industry adoption has resulted into an emerging community of developers actively contributing in the development of libraries and template smart contracts for industry-standardised tokens and use cases, development tools, and testing frameworks among others.

6.3.1 Contributed Libraries and Usage. Several template contracts have been developed in Scilla *by the community.* Those include ERC223 and ERC777—security-hardened variants of ERC20, contracts for crowdsales, escrows, a faucet contract to distribute ERC20 tokens, contracts for access control, and even the new upcoming standard ERC1404 for security tokens that will allow ownership and transfer of traditional financial assets.

Some of the most prominent decentralised applications that have been written in Scilla and are live include a blockchain-based game that uses an nft token contract to represent in-game assets and provides a marketplace to trade them; a name registry contract that allows users to register a human-readable name (*e.g.,* `myname.zil`) for their account address (*e.g.,* `0x1b133c67ae12...`) and use the human-readable name to receive payments; a hash time-locked contract to enable cross-chain atomic swaps; and a contract that analyzes tweets and rewards users who tweet a certain text. The addresses of those contracts are provided in Sec. A.1.

Other development tools that the community is involved in include the support for language server protocol (LSP 2018), a web-based IDE,[19] and a Scilla plugin for Visual Studio Code editor.[20]

6.3.2 Finding Bugs in Community Contracts. We have used the analysers from Sec. 5.1 and Sec. 5.2 on sixteen community-written contracts (*cf.* Sec. A.2) and identified the following inefficiency patterns/bugs: (a) copying of a map from a field to read a single entry, (b) copying of a nested map to check if the key corresponding to it exists, (c) creating a copy of a container and not referencing it later, (d) money-receiving transitions failing to explicitly accept funds or accepting funds more than once. The bugs from categories (a)–(c) were discovered using the resource analyser, while (d) has been detected via the cash-flow analysis.

7 DISCUSSION

This manuscript has described Scilla v1.0. Indeed, we have foreseen some future changes in the language design. Scilla comes with a mechanism of versioning to cater for this evolution, but presentation of it is outside of the scope of this paper.

Notes on the current design. The initial version of Scilla by Sergey et al. (2018a) featured explicit continuations for non-atomic calls to other contracts' transitions, in order to mimic Solidity-style programming. Having implemented contracts from Tab. 2, we observed that all interaction between contracts can be structured in a tail-call style, rendering in-transition external calls needless.[21] This allowed us to simplify the communication aspect *wrt.* to the initial proposal.

[19] https://savant-ide.zilliqa.com/

[20] https://marketplace.visualstudio.com/items?itemName=as1ndu.scilla

[21] This is consistent with the security recommendation to favour *pull-style* withdrawal in Solidity contracts.

The initial design only allowed for simple imperative interactions with the contract state (reading/writing fields). We had to extend this interface with `map`-specific statements (Fig. 1e), to provide special support for maps which are frequently used for accounting. Had we ignored this paramount use-case, most of the contract's transitions would have linear (as opposed to constant) gas use, due to the need to load/store a map for every manipulation. In contrast, with introduction of those imperative operations, the gas cost of running contracts such as ERC20 became constant (*cf.* Tab. 2).

First class polymorphic functions are essential for generic programming (Lämmel and Jones 2003) or imposing restrictions on what users of higher-order functions are allowed to do with the input of the functions they provide. This technique is crucial to ensure, *e.g.*, the safety of ST monad (Launchbury and Jones 1994). As an intermediate language, Scilla must allow for implementations of expressive higher-level languages on top of it, including the support for ad-hoc polymorphism. It is known that, *e.g.*, Haskell-like type classes cannot, in general, be implemented by a translation into the standard Hindley-Milner type system with prenex polymorphism (Jones 1997), hence our choice of System F as a sufficiently expressive type system for our purposes.

Input/output and interaction with the protocol. As revealed, the evaluation in Sec. 6.2, input/output poses a significant execution overhead. As our immediate future work, we are working on implementing an on-demand direct storage access, where the entire state is not passed around in files but only necessary parts are fetched on-demand. With that in place, we expect Scilla performance to be almost on par with that of Ethereum contracts, keeping in mind that Scilla is currently interpreted. More concretely, we expect the non-constant (increasing with state size) serialisation times in Tab. 3 to become independent of the state size.

Since Scilla evaluator interacts with the state via a set of monadic primitives, it should be easier to switch the state representation, as long as it supports the same interface of I/O primitives. For instance, *flat-buffers*[22] can be a good alternative to the current JSON-based serialisation. Flat-buffers will allow access to serialised data without the need for parsing/unpacking and hence leading to better memory efficiency and execution performance of the evaluator. However, more research needs to be done on distilling the right interface, so multiple consensus protocols could provide it, simplifying the adoption of Scilla.

Storing contract state on the blockchain. Scilla contracts will require roughly the same persistent storage as Ethereum contracts as the underlying storage model is the same. We however believe that the storage scalability issues with blockchains in general cannot be fully solved at the language-level. Instead, improvements need to be made at the protocol-level. Some of the ideas that we are currently exploring are: (a) state sharding—each node will only store a part of the global state and (b) outsourcing of persistent storage to a distributed storage provider such as IPFS.[23]

8 RELATED WORK

Many specialised programming languages for smart contracts were proposed recently by researchers and industry practitioners, aiming to either improve on EVM and Solidity (Ethereum's high-level language), or targeting a particular blockchain platform. To date, most of those languages are available in a form of a sparsely documented repository, a position paper, or a blog post (*e.g.*, Obsidian (Coblenz 2017), Babbage (Reitwiessner 2017), Marlowe (IOHK Foundation 2019a), Bamboo (Hirai 2018), Rholang (RChain Cooperative 2019), Sophia (æternity Blockchain 2019), and Ligo (Alfour 2019)), which makes it difficult to conduct a rigorous comparison. In the interest

[22] https://google.github.io/flatbuffers
[23] https://ipfs.io

of space, we only relate Scilla to proposals that come with a publicly available documentation *and*, to the best of our knowledge, have been deployed on an open blockchain protocol.

Low- and intermediate-level languages. EVM's shortcoming when it comes to tractability and unsafe features have been recognised by the community, and several alternatives were suggested.

IELE (Rosu 2018) is a low-level virtual machine, similar in design to EVM, but prohibiting some of its features, such as unguarded calls to other contracts and the `delegatecall` instruction. Yul (Ethereum Foundation 2018f) is an intermediate language developed for interoperability between high-level contract languages that build on Ethereum and several versions of EVM. Pact (Popejoy 2017) is LISP-like intermediate language, which is optimised for the database-like queries on the state managed by the Kadena blockchain protocol and used for communication.

Michelson is a low-level stack-based Turing-complete language by Tezos Foundation (2018a), which, unlike EVM, is statically typed and offers high level algebraic data types, immutable functional data structures such as lists, maps, lazily deserialised maps, and arbitrary precision arithmetic. Michelson provides a typing discipline for stack operations of the form `(type of stack before) → (type of stack after)` and ensures that no smart contract execution fails because an instruction has been executed on a stack of unexpected length or contents. For instance, the instruction `DUP` for duplicating the top of the stack has the type `DUP :: 'a : 'A →'a : 'a : 'A`. We should note at this point that Michelson's types are monomorphic: unlike Scilla it does not feature polymorphic functions.[24] The Michelson interpreter is a pure function that uses the state threading technique. Michelson contracts, unlike Scilla's, have a single entry point and take only one parameter as the top of the initial stack containing a single pair of an input value and explicitly passed state called storage. As its last execution step, a well-behaved Michelson contract returns a stack with a pair consisting of a list of internal blockchain operations and an updated state. The internal operations get queued for execution when the contract returns. Alternatively, a Michelson contract fails if the programmer explicitly calls `FAILWITH` instruction or because of a run-time error such as division by zero, gas exhaustion, etc. the type system cannot detect. To simulate Scilla's transitions the programmer can define the input parameter of a sum type and use a product type to represent multiple transition parameters. The low-level nature of Michelson (compared to the one of Scilla) makes it a more difficult target for automated analyses and lightweight verification: to the best of our knowledge, no analogues of our gas usage (Sec. 5.1) and cash-flow (Sec. 5.2) analyses exist for Michelson. We also believe that the transition-based (as opposed to stack-based) execution model for multi-contract interactions make Scilla more amenable to verification of temporal properties (Sergey et al. 2018b).

Move (Blackshear et al. 2019) 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 (Girard 1987): 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 loose 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

[24] The type variables in the type of the (second-class) `DUP` operation are merely meta-variables.

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 (Trunov 2019).

High-level languages. Ethereum's SOLIDITY is the de facto high-level programming language for smart contracts, featuring a JavaScript-like syntax. Due to its expressivity, and also constantly changing design, it is a moving target for verification and static analysis. That said, because of its popularity, a number of industrial-strength analysis frameworks for SOLIDITY have been developed recently. VYPER by Ethereum Foundation (2018e) is a Python-like language aimed to rectify some of SOLIDITY's issues by removing some the "dangerous" features, such as, *e.g.*, infinite-length loops.

LIQUIDITY by Tezos Foundation (2018b) is an ML-style language (compiled to MICHELSON), similar to SCILLA in its pure fragment. Unlike SCILLA, LIQUIDITY does not make communication between contracts explicit and allows for general recursion, making it the programmer's duty to pass the correct state to a callee contract and handle the result. We believe that choice to restrict effectful inter-contract interaction to communication we made in SCILLA has made it simple (if not possible) to analyse contracts (*e.g.*, for gas and cash-flows as in Sec. 5) in isolation. Since no similar analyses are available for LIQUIDITY, no direct comparison is possible.

FLINT by Schrans et al. (2018) is a type-safe SOLIDITY-like language, which can be compiled down to EVM via YUL. FLINT relies on a substructural typing discipline for addressing the issues similar to those we outlined in Sec. 2. It uses object capabilities for avoiding DAO-like exploits, limiting the power of a callee contract to modify the caller's state, and also employs a special asset type `Wei` to account for cash-flows (Schrans 2018). For the same purposes, SCILLA uses automata-based contract structure and a cash-flow analysis (which, unlike the asset type, allows for non-native tokens).

Low-level languages for UTXO scripting. While SCILLA has been designed to work with *account-based* model for blockchain state (adopted, for instance, by Ethereum), a number of languages exist for an alternative, UTXO transaction model (Sun 2018), adopted in particular by Bitcoin and Cardano blockchains. The most prominent languages include Bitcoin Script (Bitcoin Wiki 2017) and SIMPLICITY (O'Connor 2017) (for Bitcoin), and PLUTUS CORE (IOHK Foundation 2019b) (for Cardano). Due to the absence of explicit state, the UTXO model allows for a simpler, purely-functional design of a language (*e.g.*, PLUTUS is a strict subset of Haskell), while making it more difficult to express stateful computations, which are needed, *e.g.*, for persistent accounting.

9 FUTURE DIRECTIONS AND CONCLUSION

A few important features are still missing from the currently deployed version of SCILLA. Our immediate plan is to add support for *recursive* user-defined data types, with the corresponding recursion principles (folds). We are also planning to develop a family of higher-level languages, which translate down to SCILLA, to support most common applications, such as token-based games.

Looking ahead, SCILLA opens exciting opportunities for bringing the state-of-the art research in PL and formal methods into the emerging area of programmable consensus protocols. In the near future, we are interested in building a shallow embedding of SCILLA into the Coq proof assistant, allowing one to reason about safety and temporal contract properties. A further agenda is to implement the reference evaluator for SCILLA directly in a proof assistant. Finally we are going to leverage the analyses from Sec. 5 for SMT-based verification of most common contract invariants, their safety, and temporal properties (Sergey et al. 2018b).

The field of smart contract programming is still relatively young, but is desperately in need for firm formal foundations, which would enable principled reasoning about programs, their analysis and verification. We believe that our design of SCILLA is a logical step towards achieving this goal.

A NOTABLE SCILLA CONTRACTS

In this appendix, we provide addresses and links to some noteworthy Scilla contracts contributed by the community members.

A.1 Contributed Contracts on Zilliqa Main-Net

The following are some representative third-party Scilla contracts, mentioned in Sec. 6.3.1 and available on Zilliqa blockchain:

- A blockchain-based game of collectibles:
  - Game contract: zil1vxl33hrua4wsld32zk2fjm6qv3qu4tg6cw4azu
  - NFT marketplace contract: zil1w0gj7tnxk8usu68et44jfchuwh0mjc040pqd6l
- A name registry: zil1jcgu2wlx6xejqk9jw3aaankw6lsjzeunx2j0jz
- A contract for cross-chain atomic swaps: zil1j2vvhfx783kchv70heac2tp0er8772c7dw3wa4
- A contract rewarding users for tweets: zil15pv5kyhk767sgvzp0u79gjl5a48e290m409r5n

A.2 Community Contract Repositories

The contracts for analysis case studies in Sec. 6.3.2 were taken from the following repositories:

- Simple Reusable Vanilla Contracts for Scilla: https://github.com/merkaliser/scilla-vanilla
- Decentralised exchange for fungible tokens on Zilliqa: https://github.com/khelmy/zdex

ACKNOWLEDGMENTS

First and foremost we wish to thank the amazing Zilliqa team, whose involvement helped a lot to shape up the design and implementation of Scilla, and deploy it on the main-net blockchain. In particular, we thank Sandip Bhoir, Han Wen Chua, Sophia Fang, Deli Gong, Sheng Guang Xiao, Yaoqi Jia, Saiba Kataruka, Edison Lim, Haichuan Liu, Antonio Nicolas Nunez, Advay Pal, Kaustubh Shamshery, Hugh Sipiere, Bryan Tan Yao Hong, Jun Hao Tan, Ian Tan, Clark Yang, and Noel Yoo. We are thankful to Xinshu Dong, Aquinas Hobor, Prateek Saxena, Max Kantelia, and Juzar Motiwalla for their encouragement and support of the Scilla initiative. We feel very grateful to the vibrant community of Zilliqa early adopters, whose feedback and technical contributions were instrumental for adapting the initial design of Scilla to make it more development- and verification-friendly.

We benefited a lot from the discussions on Scilla design with members of the programming languages and verification community: Mario Alvarez, Olivier Danvy, Dominique Devriese, Sophia Drossopoulou, Fritz Henglein, Ranjit Jhala, Neel Krishnaswami, Ben Livshits, Sukyoung Ryu, Mooly Sagiv, Bas Spitters, and Petar Tsankov. We thank the OOPSLA 2019 PC and AEC referees for their careful reading and many constructive suggestions on the paper and the implementation.

Finally, we thank the sponsors of the Crystal Centre at the School of Computing of National University of Singapore that has supported Ilya Sergey's research.

REFERENCES

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

Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. 2008. Automatic Inference of Upper Bounds for Recurrence Relations in Cost Analysis. In *SAS (LNCS)*, Vol. 5079. Springer, 221–237.

Gabriel Alfour. 2019. Introducing LIGO: a new smart contract language for Tezos. https://medium.com/tezos/introducing-ligo-a-new-smart-contract-language-for-tezos-233fa17f21c7.

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

Leonardo Alt and Christian Reitwießner. 2018. SMT-Based Verification of Solidity Smart Contracts. In *ISoLA (LNCS)*, Vol. 11247. Springer, 376–388.

Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. 2018. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *CPP*. ACM, 66–77.

Nada Amin and Tiark Rompf. 2017. Type soundness proofs with definitional interpreters. In *POPL*. ACM, 666–679.

Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A Survey of Attacks on Ethereum Smart Contracts (SoK). In *POST (LNCS)*, Vol. 10204. Springer, 164–186.

Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2017. Consensus in the Age of Blockchains. *CoRR* abs/1711.03936 (2017).

Kshitij Bansal, Eric Koskinen, and Omer Tripp. 2018. Automatic Generation of Precise and Useful Commutativity Conditions. In *TACAS (LNCS)*, Vol. 10805. Springer, 115–132.

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. 2016. Formal Verification of Smart Contracts: Short Paper. In *PLAS*. ACM, 91–96.

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

Sam Blackshear, Evan Cheng, David L. Dill, Victor Gao, Ben Maurer, Todd Nowacki, Alistair Pott, Shaz Qadeer, Rain, Dario Russi, Stephane Sezer, Tim Zakian, and Runtian Zhou. 2019. Move: A Language With Programmable Resources. https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf.

Jialiang Chang, Bo Gao, Hao Xiao, Jun Sun, and Zijiang Yang. 2018. sCompile: Critical Path Identification and Analysis for Smart Contracts. *CoRR* abs/1808.00624 (2018).

Arthur Charguéraud. 2013. Pretty-Big-Step Semantics. In *ESOP (LNCS)*, Vol. 7792. Springer, 41–60.

Ting Chen, Xiaoqi Li, Xiapu Luo, and Xiaosong Zhang. 2017. Under-optimized smart contracts devour your money. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER*. IEEE Computer Society, 442–446.

Michael Coblenz. 2017. Obsidian: A Safer Blockchain Programming Language. In *ICSE (Companion)*. IEEE Press, 97–99.

Coq Development Team. 2019. *The Coq Proof Assistant Reference Manual - Version 8.9*. http://coq.inria.fr.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. ACM, 238–252.

Olivier Danvy. 2019. Folding left and right over Peano numbers. *J. Funct. Program.* 29 (2019), e6.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. 151–160.

Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391.

Olivier Danvy and J. Michael Spivey. 2007. On Barron and Strachey's cartesian product function. In *ICFP*. ACM, 41–46.

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

Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muslu, and Todd W. Schiller. 2011. Building and using pluggable type-checkers. In *ICSE*. ACM, 681–690.

Ethereum Foundation. 2018a. Decentralized Autonomous Organization. https://www.ethereum.org/dao.

Ethereum Foundation. 2018b. ERC20 Token Standard. https://theethereum.wiki/w/index.php/ERC20_Token_Standard.

Ethereum Foundation. 2018c. List of Known Solidity Bugs. https://solidity.readthedocs.io/en/v0.5.7/bugs.html, accessed on Apr 5, 2019.

Ethereum Foundation. 2018d. Solidity Documentation. http://solidity.readthedocs.io.

Ethereum Foundation. 2018e. Vyper. https://vyper.readthedocs.io.

Ethereum Foundation. 2018f. Yul. https://solidity.readthedocs.io/en/latest/yul.html.

Andrzej Filinski. 1994. Representing Monads. In *POPL*. ACM Press, 446–457.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247.

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102.

Jean-Yves Girard. 1972. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. Dissertation. Université Paris 7.

Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. 2018. MadMax: surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL* 2, OOPSLA (2018), 116:1–116:27.

Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A Semantic Framework for the Security Analysis of Ethereum Smart Contracts. In *POST (LNCS)*, Vol. 10804. Springer, 243–269.

Shelly Grossman, Ittai Abraham, Guy Golan-Gueta, Yan Michalevsky, Noam Rinetzky, Mooly Sagiv, and Yoni Zohar. 2018. Online detection of effectively callback free objects with applications to smart contracts. *PACMPL* 2, POPL (2018).

Emin Gün Sirer. 2016. Reentrancy Woes in Smart Contracts. http://hackingdistributed.com/2016/07/13/reentrancy-woes/

Robert Harper. 2012. *Practical Foundations for Programming Languages*. Version 1.32.

Yoichi Hirai. 2018. Bamboo. https://github.com/pirapira/bamboo.

Jan Hoffmann, Ankush Das, and Shu-Chun Weng. 2017. Towards automatic resource bound analysis for OCaml. In *POPL*. ACM, 359–373.

IOHK Foundation. 2019a. Marlowe: A Contract Language For The Financial World. https://testnet.iohkdev.io/marlowe/.

IOHK Foundation. 2019b. Plutus: A Functional Contract Platform. https://testnet.iohkdev.io/plutus/.

Mark P. Jones. 1997. First-class Polymorphism with Type Inference. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 483–496.

Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. Zeus: Analyzing Safety of Smart Contracts. In *NDSS*.

Andrew Kennedy. 1997. Relational Parametricity and Units of Measure. In *POPL*. ACM Press, 442–455.

Aashish Kolluri, Ivica Nikolić, Ilya Sergey, Aquinas Hobor, and Prateek Saxena. 2018. Exploiting The Laws of Order in Smart Contracts. *CoRR* abs/1810.11605 (2018). arXiv:1810.11605

Johannes Krupp and Christian Rossow. 2018. teEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts. In *USENIX Security Symposium*. USENIX Association, 1317–1333.

Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. 2014. CakeML: a verified implementation of ML. In *POPL*. ACM, 179–192.

Ralf Lämmel and Simon Peyton Jones. 2003. Scrap your boilerplate: a practical design pattern for generic programming. In *Proceedings of the 2003 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*. ACM, 26–37.

John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*. ACM, 24–35.

James J. Leifer, Gilles Peskine, Peter Sewell, and Keith Wansbrough. 2003. Global abstraction-safe marshalling with hash types. In *ICFP*. ACM, 87–98.

Sheng Liang, Paul Hudak, and Mark P. Jones. 1995. Monad Transformers and Modular Interpreters. In *POPL*. ACM Press, 333–343.

LSP 2018. Language Server Protocol. https://langserver.org.

Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making Smart Contracts Smarter. In *CCS*. ACM, 254–269.

Nancy A. Lynch and Mark R. Tuttle. 1989. An Introduction to Input/Output Automata. *CWI Quarterly* 2 (1989), 219–246.

Matteo Marescotti, Martin Blicha, Antti E. J. Hyvärinen, Sepideh Asadi, and Natasha Sharygina. 2018. Computing Exact Worst-Case Gas Consumption for Smart Contracts. In *ISoLA (LNCS)*, Vol. 11247. Springer, 450–465.

Yaron Minsky. 2016. Let syntax, and why you should use it. Blog post, available at https://blog.janestreet.com/let-syntax-and-why-you-should-use-it.

John C. Mitchell. 2003. *Concepts in programming languages.* Cambridge University Press.

Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

Satoshi Nakamoto. 2008. Bitcoin: A peer-to-peer electronic cash system. Abailable at http://bitcoin.org/bitcoin.pdf.

Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. 2018. Finding The Greedy, Prodigal, and Suicidal Contracts at Scale. In *ACSAC*. ACM. To appear.

Russell O'Connor. 2017. Simplicity: A New Language for Blockchains. https://blockstream.com/simplicity.pdf.

Scott Owens, Magnus O. Myreen, Ramana Kumar, and Yong Kiam Tan. 2016. Functional Big-Step Semantics. In *ESOP (LNCS)*, Vol. 9632. Springer, 589–615.

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

Simon L. Peyton Jones. 2013. Type-Directed Compilation in the Wild: Haskell and Core. In *TLCA (LNCS)*, Vol. 7941. Springer.

George Pîrlea and Ilya Sergey. 2018. Mechanising Blockchain Consensus. In *CPP*. ACM, 78–90.

Robert Pollack. 1990. Implicit Syntax. In *Informal Proceedings of First Workshop on Logical Frameworks, Antibes.*

Stuart Popejoy. 2017. The Pact Smart-Contract Language, Revision 1.5. http://kadena.io/docs/Kadena-PactWhitepaper.pdf.

RChain Cooperative. 2019. Rholang. https://rholang.rchain.coop.

Christian Reitwiessner. 2017. Babbage—a mechanical smart contract language. Online blog post.

John C. Reynolds. 1974. Towards a theory of type structure. In *Programming Symposium (LNCS)*, Vol. 19. Springer, 408–423.

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

Michael Rodler, Wenting Li, Ghassan O. Karame, and Lucas Davi. 2019. Sereum: Protecting Existing Smart Contracts Against Re-Entrancy Attacks. In *NDSS*.

Grigore Rosu. 2018. IELE: A New Virtual Machine for the Blockchain. https://iohk.io/blog/iele-a-new-virtual-machine-for-the-blockchain.

Franklin Schrans. 2018. *Writing Safe Smart Contracts in Flint.* Master's thesis. Imperial College London, Department of Computing.

Franklin Schrans, Susan Eisenbach, and Sophia Drossopoulou. 2018. Writing safe smart contracts in Flint. In *<Programming> (Companion)*. ACM, 218–219.

Ilya Sergey, Dominique Devriese, Matthew Might, Jan Midtgaard, David Darais, Dave Clarke, and Frank Piessens. 2013. Monadic Abstract Interpreters. In *PLDI*. ACM, 399–410.

Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018a. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR* abs/1801.00687 (2018). http://arxiv.org/abs/1801.00687

Ilya Sergey, Amrit Kumar, and Aquinas Hobor. 2018b. Temporal Properties of Smart Contracts. In *ISoLA (LNCS)*, Vol. 11247. Springer, 323–338.

Ilya Sergey, Dimitrios Vytiniotis, and Simon L. Peyton Jones. 2014. Modular, higher-order cardinality analysis in theory and practice. In *POPL*. ACM, 335–348.

Peter Sestoft. 1996. ML Pattern Match Compilation and Partial Evaluation. In *International Seminar on Partial Evaluation, Dagstuhl Castle (LNCS)*, Vol. 1110. Springer, 446–464.

Amin Shali and William R. Cook. 2011. Hybrid Partial Evaluation. In *OOPSLA*. ACM, 375–390.

Dieter Shirley. 2018. ERC-721. http://erc721.org/.

Jeremy Siek. 2012. Big-step, diverging or stuck? http://siek.blogspot.com/2012/07/big-step-diverging-or-stuck.html.

Jeremy Siek. 2013. Type safety in three easy lemmas. http://siek.blogspot.com/2013/05/type-safety-in-three-easy-lemmas.html.

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

Nick Szabo. 1994. Smart Contracts. Online manuscript.

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

Tezos Foundation. 2018b. Liquidity. http://www.liquidity-lang.org/.

Sergei Tikhomirov, Ekaterina Voskresenskaya, Ivan Ivanitskiy, Ramil Takhaviev, Evgeny Marchenko, and Yaroslav Alexandrov. 2018. SmartCheck: Static Analysis of Ethereum Smart Contracts. In *WETSEB@ICSE*. ACM, 9–16.

Anton Trunov. 2019. A Scilla vs Move case study. Blog post available at https://medium.com/@anton_trunov/a-scilla-vs-move-case-study-afa9b8df5146.

Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical Security Analysis of Smart Contracts. In *CCS*. ACM, 67–82.

Peng Wang. 2019. *Type System for Resource Bounds with Type-Preserving Compilation*. Ph.D. Dissertation. Massachusetts Institute of Technology.

Gavin Wood. 2014. Ethereum: A Secure Decentralized Generalised Transaction Ledger.

Andrew K. Wright and Matthias Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94.

Zilliqa Team. 2017. The Zilliqa Technical Whitepaper. https://docs.zilliqa.com/whitepaper.pdf Version 0.1.