Contents lists available at ScienceDirect

Blockchain: Research and Applications

journal homepage: www.journals.elsevier.com/blockchain-research-and-applications

Analysis of smart contracts balances

Cosimo Laneve, Claudio Sacerdoti Coen *

Department of Computer Science and Engineering, University of Bologna, INRIA Focus, Bologna, 40126, Italy

A R T I C L E I N F O

Keywords:
Smart contracts
Blockchain
Static analysis
Cost analysis
Cryptocurrency
Movements

A B S T R A C T

We define a technique for analyzing updates of smart contracts balances due to transfers of digital assets. The analysis addresses a lightweight smart contract language and consists of a two-step translation. First, we define the input-output behaviors of smart contract functions by means of a simple functional language with static dispatch. Then we associate the terms of this intermediate language with cost equations that compute the loss or gain of digital assets. The resulting equations can be fed to an off-the-shelf cost analyzer to provide upper bounds to the loss or gain. Our analysis has been prototyped and we report its assessments and discuss extensions with additional features.

1. Introduction

Smart contracts are programs that run on distributed networks with nodes storing a common state in the form of a blockchain. These programs are gaining more and more interest because they implement applications that can manage and transfer assets of considerable value (usually in the form of cryptocurrencies, like Bitcoin), called decentralized applications. Examples of such applications are food supply chain management, energy market management, and identity notarization.

Several smart contracts languages have been recently proposed for specifying decentralized applications, such as the Bitcoin Scripting [1], Solidity for Ethereum [2], Move for Libra [3]. Security guarantees in these languages are of paramount importance because it is possible to program the transfer of large capitals. Actually, already in the past few years, several millions of USD have been lost because of subtle flaws in the smart contracts [4,5].

To alleviate the burden of smart contract analysis, a number of automated techniques have been designed for verifying relevant properties, such as liquidity [6], gas consumption [7], and compliance and violation of programming patterns [8]. This contribution follows these lines of research by focussing on another critical feature that is at the core of famous attacks: the transfer of cryptocurrency assets from one smart contract to another. Indeed, our technique automatically computes the upper bounds of the amount of cryptocurrency gained and lost by a smart contract during a transaction.

We carry this study on a language for smart contracts whose constructs have been inspired by Solidity. The language is lightweight because it does not have complex features such as new contracts instantiation, inheritance, try-catch exception handling, arrays, and mappings. In our setting, programs are a (finite) set of smart contracts whose functions may either update the state or transfer cryptocurrencies or abort or invoke other functions. Overall, our model is simple and rigorous, which are, in our opinion, fundamental criteria for reasoning about the properties of smart contracts and for understanding their basic principles. Once the properties on the core model have been analyzed, one can address other, more complex features that are drawn from the mainstream smart contract languages.

1.1. Our contribution

In Section 2, we define `mSCL`, an acronym for mini Smart Contract Language, which is dubbed minuscule. `mSCL` has function invocations, field updates, conditional behavior, cryptocurrency transfer, fallback functions, recursion, and failures. More importantly, `mSCL` has a formal operational semantics that is defined in Section 3 and is expressive enough to define standard attacks, cf. the Bank-Thief contracts in Example 1.

The transfer of digital assets between `mSCL` smart contracts is analyzed by means of cost analyzers [9,10]. These cost analyzers use (declarative) languages that have a rigid structure and poor expressivity. For example, predicates must be written in disjunctive normal form, data types are only numbers (integers and reals) and cost functions are stateless. Encoding in these rigid languages the intricacies of the semantics of transfers of assets and of failures, the states of smart contracts, environments, and boolean expressions turns out to be painful and ad-hoc. For this reason, we decided to separate semantics concerns of

* Corresponding author.
E-mail address: claudio.sacerdoticoen@unibo.it (C. Sacerdoti Coen).

https://doi.org/10.1016/j.bcra.2021.100020
Received 18 December 2020; Received in revised form 7 May 2021; Accepted 14 June 2021

mSCL from concerns due to the rigidity of cost analyzers and to the definition of the appropriate cost model. Therefore, in Section 4, we introduce an intermediate functional language that has static dispatch and admits environments as primitive data types, tail function invocations, and every predicate and operator of Presburger arithmetics. Functions in the intermediate language are intended to define the input-output behavior of mSCL functions. Indeed, they take in input environments (that model the state of programs) and return environments. (Other intermediate languages have been defined for smart contracts; a thorough discussion is reported in Section 8.)

While the intermediate language has a very simple operational semantics (two rules only), which allows us to establish the correctness of the translation in a standard way, it is inadequate to express mSCL functionalities in a direct way. In particular, a number of mSCL features have required an explicit encoding that entangles the translation: failures and the corresponding definition of backtracking, implicit and explicit management of assets (such as the complexities due to the fallback), function invocations with explicit continuations (which should have required a higher-order language for expressing the Continuation Passing Style, while the intermediate language is first-order).

In Section 5, we derive cost equations from terms in the intermediate language. These equations are associated with two cost models: one for computing the loss of a smart contract at the end of a transaction and the other for the gain. To this aim, we flatten the environments (in order to feed cost functions with tuples of values), normalize the predicates, and select adequate cost models. It is worth noticing that the normalization process gives an exponential number of equations with respect to the equations without normalized predicates: our analysis would benefit by a cost analyzer that accepts generic predicates. The normalized cost equations can be fed to a cost analyzer to compute upper bounds to the loss and gain.

We have prototyped our technique and run the prototype on several smart contracts that have been downloaded from Ethereum (and adapted to mSCL). Section 6 reports the assessments obtained by running the prototype on a few archetypal examples (a Bank-Thief code, an English Auction Scheme, and two Ponzi Schemas). These examples have been chosen to highlight the issues of our technique and the current prototype. We notice that, because of scalability problems, our analysis can be profitably used only in presence of aggressive optimizations. As discussed in the conclusions, this is a matter of current research.

In Section 7 we also study an extension of mSCL with additional features (that are also inspired by Solidity). In particular, the extension of mSCL function invocations with explicit continuations allows us to express famous attacks, such as the DAO [4]. We discuss the related works in Section 8 and we deliver our conclusions in Section 9.

1.2. *Captatio benevolentiae*

This work is not intended to address Solidity and to provide a full-fledged analyzer for that language (which is an industrial project that would require a different effort). It is rather a proof-of-concept about how to compute the cryptocurrency movements in generic smart contract languages. Solidity has been used as an inspiration source to design our mSCL language that models, we hope, the innovative features of smart contracts (transfer of cryptocurrencies, and ACID properties of transactions obtained by reverting to the initial state in case of errors).

A key contribution of the paper is the definition of the intermediate language and the development of the analysis technique for it. Once the back-end of the analyzer for the intermediate language is in place, it will be sufficient to define the translation of any source code into the intermediate code for verifying the corresponding updates of balances. In our mind, the intermediate language is a decoupling point between front-ends that deal with different smart contract languages and back-ends that apply different techniques to analyze the code. Actually, our intermediate language, being much simpler than the source language, may be equipped with several analyses, in such a way that verifying a source language amounts to compiling it to the intermediate one (thus forgetting about all the technicalities of the analysis). For example, it is possible to reuse analyses such as computational cost and gas consumption since the corresponding techniques have been already developed for the target cost equation language [7,10].

2. The mSCL calculus

The mini Smart Contract Language, noted mSCL and dubbed minuscule, is a calculus featuring a minimal set of smart contract primitives, such as function invocations, field updates, conditional behavior, cryptocurrency transfer, recursion, and failures that are inspired to Solidity.

We use a countable set of smart contract names *Id*, ranged over by C, D, H, a countable set of function names, ranged over by $\mathtt{m}$, $\mathtt{m}'$, a countable set of field names *FId*, ranged over by $\mathtt{f}$, $\mathtt{f}'$, and a countable set *Var* of variables, ranged over by x, y, z. Variables include field names and smart contract names.

The syntax of mSCL is

$$
\begin{array}{rl}
\mathsf{C} ::= & \mathtt{contract}\ C\ \{\ \overline{\mathsf{T}\ \mathtt{f};}\ \ \mathsf{F}\ \left[\mathtt{fallback(\,)\ payable}\ \{\ \}\right]\ \} \\[4pt]
\mathsf{F} ::= & \varepsilon \mid \mathtt{function}\ \mathtt{m}(\overline{\mathsf{T}\ x})\ [\mathtt{payable}]\ \{\overline{\mathsf{T}\ z};\ \mathsf{S}\}\ \mathsf{F} \\[4pt]
\mathsf{T} ::= & \mathtt{uint} \\[4pt]
\mathsf{S} ::= & \varepsilon \mid x{=}\mathsf{E};\ \mathsf{S} \mid \mathtt{if}\ (\mathsf{E})\ \{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}\ \} \mid \mathsf{E}.\mathtt{m}[.\mathtt{value}(\mathsf{E})](\overline{\mathsf{E}}); \\
& \mid \mathtt{revert};\mid \mathsf{E}.\mathtt{transfer}(\mathsf{E});\ \mathsf{S} \\[4pt]
\mathsf{E} ::= & n \mid x \mid \mathtt{this} \mid \mathsf{E} \sharp \mathsf{E} \mid\ !\mathsf{E} \mid \mathtt{msg.sender} \\
& \mid \mathtt{msg.value} \mid \mathsf{E}.\mathit{balance} \\[4pt]
\sharp ::= & + \mid - \mid > \mid = \mid \geqslant \mid \&\& \mid * \mid /
\end{array}
$$

where terms written within "[" and "]" are optional. An mSCL program $\mathcal{P}$ is a sequence of smart contract definitions $(\mathsf{C}_1, \cdots, \mathsf{C}_n)$, which, in turn, are sequences of fields and function definitions. If $\mathsf{C} = \mathtt{contract}\ C\ \{\cdots\}$, we say that C is the smart contract name of C and we address the set of contract names of $\mathcal{P}$ with *cnames*$(\mathcal{P})$.

In a contract $\mathtt{contract}\,C\{\ \overline{\mathsf{T}\,\mathtt{f};}\,\mathsf{F}\ [\mathtt{fallback()\ payable}\{\}]\ \}$, the fields are $\overline{\mathsf{T}\,\mathtt{f};}$ and the corresponding set is *fields*(C), the functions are either those in F or the $\mathtt{fallback}$. We write $\mathtt{m}(\overline{\mathsf{T}x})[\mathtt{payable}]\{\overline{\mathsf{T}z};\ \mathsf{S}\} \in C$ if the function belongs to the contract named C and similarly for $\mathtt{fallback} \in C$. Additionally, in function m $\mathtt{function}\,\mathtt{m}(\overline{\mathsf{T}x})[\mathtt{payable}]\{\overline{\mathsf{T}z};\ \mathsf{S}\}$, $\overline{\mathsf{T}x}$ are the formal parameters and $\overline{\mathsf{T}z}; \mathsf{S}$ is the body of $\mathtt{m}$, where $\overline{\mathsf{T}z}$ are the local variables. We assume that fields, formal parameters, and local variables do not contain duplicate names.

Smart contracts have an implicit field—the balance—that records the cryptocurrency stored in the contract. This field is updated either (i) when a payable function is invoked (in this case the balance is increased by the cryptocurrencies carried by the invocation—keyword value), or (ii) when the cryptocurrency is explicitly transferred (the operation transfer).

The fallback function, when present, allows a contract to accept cryptocurrency transfers. In particular, the transfer of cryptocurrencies also includes the invocation of the callee's fallback function (The semantics of transfer in Fig. 2 does not model this invocation of fallback because the corresponding body is always empty.). If the callee has no fallback then cryptocurrency transfers to it are refused and always backtrack. Similarly, since in mSCL the invocations of the undeclared functions default to the fallback function, when it misses, a backtrack occurs. In these cases, the fallback function ignores all actual parameters of an undeclared function, except the transferred cryptocurrencies.

Statements S include the empty statement ε; the assignment $x = \mathsf{E}$ followed by a continuation, where x may be either a field or a formal parameter or a local variable; conditionals; the invocation of a function in the two formats $\mathsf{E}.\mathtt{m}(\overline{\mathsf{E}'})$ and $\mathsf{E}.\mathtt{m}.\mathtt{value}(\mathsf{E}'')(\overline{\mathsf{E}'})$, where E is the callee

```
contract Bank {
    fallback() payable{}
    function pay(uint n) payable{
        if (msg.value⩾ 1 && this.balance>n && n<5) {
                msg.sender.transfer(n) ;
                msg.sender.ack() ;
          }
    }
}
contract Thief {
    fallback() payable{}
    function ack() {
          msg.sender.pay.value(1)(2) ;
    }
}
```

Fig. 1. The contracts `Bank` and `Thief` in `mSCL`.

contract, `m` is the function and $\overline{\mathsf{E}'}$ are the actual parameters; the term `value`(E'') highlights when a cryptocurrency transfer occurs from the caller to the callee during the invocation (`mSCL` function invocations are external in Solidity terminology). Statements may also be `revert` that backtracks the computation to the initial store, and E.`transfer`(E') that transfers E' cryptocurrencies from the caller to E, provided caller's balance is sufficient and the callee has a fallback function (otherwise a backtrack occurs).[1]

Expressions are standard ones, except for three terms: `msg.sender` that returns the caller, `msg.value` that returns the transmitted cryptocurrencies during the invocation (to be used only inside a payable function), E.*balance* that returns the contract's balance. In the following, we use u, v to range over constant expressions or elements in *Id*.

The initial state of an `mSCL` program is determined by (i) defining the balances of the smart contracts therein, (ii) invoking a function, and (iii) specifying the caller of the invocation in (ii). See the following Example 1 for a possible initial state and Section 3 for a formal definition. It is worth noticing that the caller in (iii) may also be external (for example, it may be a smart contract that is not in the program or a user). In this case, the semantics is completely determined as long as the program does not access to its functions—with `msg.sender` (otherwise we need to make assumptions on the external caller, e.g. it must have a fallback function). Our technique will admit external callers.

Assumption 1. (Programs are typed) In the rest of the paper, we assume all `mSCL` programs to be well-typed with respect to a completely standard type system where all functions are first-order and the only two types are `uint` and `address`. Local variables and function parameters are typed by `uint` and the only expressions typed by `address` are `msg.sender`, `this`, and the names of the smart contracts defined in the program. In particular, the type systems ensure that all variables are declared before their use, that functions are only used totally applied, and that the receiver of `transfer` and function calls are only expressions of type `address`.

The features of `mSCL` are illustrated by discussing an example.

Example 1. Fig. 1 reports the codes of the contracts `Bank` and `Thief`, implementing respectively a shared bank account and a greedy client. `Bank` is used for paying clients: it has a balance and, as soon as a client invokes `pay` with a non-negative integer `n` and the balance is large enough (line 4), it withdraws `n` cryptocurrencies and transfers them to the client (line 5). In order to allow several clients to withdraw at the same time, the `Bank` only allows drawing out at most `5` cryptocurrencies for every transaction. That is, to achieve fairness between the owners of the shared account, the programmer constrains clients that want to withdraw more consistent amounts to issue multiple invocations of the `pay` function.

However, thanks to re-entrancy, `Thief` finds a way to bypass the check and grab all the money at once using just one transaction. In particular, the function `pay` also acknowledges the withdraw by invoking the client's function `ack` (because the client has paid 1 cryptocurrency for it) – line 6. This apparently harmless operation is at the core of the attack because the `ack` function of `Thief` calls back `pay` and the process continues till the account is emptied (e.g. the boolean expression at line 4 becomes false). The invocation `Bank.pay.value(1)(2)` performed by `Thief` expresses the attack.

We notice that our forthcoming technique allows one to replace the constant values 1 and 2 in Example 1 with two variables x and y, and to analyze which instances of x and y cause the attack.

2.1. Remark

There are two features that are not modeled in `mSCL`. First, nonempty fallback bodies. The analysis of this extension requires the management of explicit continuations of `transfer`, which is difficult and makes more complex the technical development of the analysis. We have preferred to deal with nonempty fallback bodies in the later Section 7 where, we hope, the analysis has been digested for the simpler setting.

Second, we do not address dynamic contract creation and deployment. In particular, we use symbolic names for smart contracts that represent smart contract addresses. When we need to model several instances of a smart contract, we simply duplicate the code, using different names. Initially, a contract knows the names of other contracts it wants to interact with, but he can also become aware of additional names later (e.g. reading `msg.sender`). This restriction allows us to avoid dependencies from the context and augment precision of the cost analysis. In Section 7 we discuss to what extent this limitation may be relaxed.

3. The semantics of `mSCL`

We use memories, ranged over ℓ, ℓ', $\cdots$, which are maps $FId \cup Var \to \mathbb{N}$. The following auxiliary functions are used in the semantic rules:

- $\ell[\mathsf{f} \mapsto v]$ is the memory update, namely $(\ell[\mathsf{f} \mapsto v])(\mathsf{f}) = v$ and $(\ell[\mathsf{f} \mapsto v])(\mathsf{g}) = \ell(\mathsf{g})$, when $\mathsf{g} \neq f$.
- $[\![e]\!]_{C',v,C,\ell}$ is a function that returns the value of e assuming C be the current contract, v be the value that has been transmitted during the invocation, C' be the caller and ℓ be the memory of C where values of fields and variables occurred in e are stored. We omit the definition of $[\![e]\!]_{C',v,C,\ell}$, which is completely standard, but we notice that the function is total thanks to the `mSCL` constraint that, in a division, the second argument is always a non-null constant. $[\![\overline{e}]\!]_{C',v,C,\ell}$ returns the tuple of values of $\overline{e}$.

A state of an `mSCL` program $\mathcal{P}$, ranged over by $\mathcal{S}$, $\mathcal{S}'$, $\cdots$, is defined by the following syntax:

$$\mathcal{S} \ ::= \quad \prod_{i\in I} C_i(\ell_i \cdot \ell_i') \mid C' \overset{v}{\blacktriangleright} C : \mathsf{S} \quad \Big| \quad \prod_{i\in I} C_i(\ell_i \cdot \ell_i') \mid 0$$

where $\prod_{i\in I} C_i(\ell_i \cdot \ell_i')$ is a parallel composition of (runtime) contracts and either $C' \overset{v}{\blacktriangleright} C : \mathsf{S}$ or 0 is the runtime statement. As usual, parallel composition in states is associative and commutative.

Runtime contracts have pairs of memories $\ell \cdot \ell'$ where ℓ is the current memory and ℓ' is the backtrack memory. The memory ℓ' is the one at the beginning of the current transaction; ℓ is a working copy of ℓ', which is updated during the transaction and it is committed if the transaction ends successfully, becoming the new backtrack memory. When we write $C(\ell \cdot$

[1] In smart contract languages, such as Solidity, actions consume gas and this gas is never returned during the backtracking. In this paper we are overlooking gas consumption since bounding gas consumption is already a well understood problem in the literature (see Ref. [2], for instance).

[EMP]
$$C' \overset{v}{\blacktriangleright} C : \varepsilon \xrightarrow{\checkmark} 0$$

[REVERT]
$$C' \overset{v}{\blacktriangleright} C : \mathtt{revert} \xrightarrow{\mathsf{fail}} 0$$

[UPD]
$$\frac{\llbracket e \rrbracket_{C',v,C,\ell} = v'}{C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : x{=}e;\mathsf{S} \longrightarrow C(\ell[x \mapsto v'] \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : \mathsf{S}}$$

[IF-TRUE]
$$\frac{\llbracket e \rrbracket_{C',v,C,\ell} \neq 0}{\begin{array}{c} C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : \mathtt{if}\ (e)\ \{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}'\ \} \\ \longrightarrow C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : \mathsf{S} \end{array}}$$

[IF-FALSE]
$$\frac{\llbracket e \rrbracket_{C',v,C,\ell} = 0}{\begin{array}{c} C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : \mathtt{if}\ (e)\ \{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}'\ \} \\ \longrightarrow C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : \mathsf{S}' \end{array}}$$

[TRANSFER]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C'' \quad \llbracket e' \rrbracket_{C',v,C,\ell} = v' \\ \ell(balance) \geqslant v' \quad \mathtt{fallback} \in C'' \\ \ell_1 = \ell[balance \mapsto^{-} v] \quad \ell_2 = \ell''[balance \mapsto^{+} v] \end{array}}{\begin{array}{c} C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{transfer}(e');\ \mathsf{S} \\ \longrightarrow C(\ell_1 \cdot \ell') \mid C''(\ell_2 \cdot \ell''') \mid C' \overset{v}{\blacktriangleright} C : \mathsf{S} \end{array}}$$

[TRANSFER-FAIL]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C'' \quad \llbracket e' \rrbracket_{C',v,C,\ell} = v' \\ \big(\ell(balance) < v' \quad \text{or} \quad \mathtt{fallback} \notin C''\big) \end{array}}{\begin{array}{c} C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{transfer}(e');\ \mathsf{S} \\ \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid C''(\ell''' \cdot \ell''') \mid 0 \end{array}}$$

[TRANSFER-SELF]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C \quad \llbracket e' \rrbracket_{C',v,C,\ell} = v' \\ \ell(balance) \geqslant v' \quad \mathtt{fallback} \in C \end{array}}{C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{transfer}(e');\ \mathsf{S} \longrightarrow C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : \mathsf{S}}$$

[TRANSFER-SELF-FAIL]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C \quad \llbracket e' \rrbracket_{C',v,C,\ell} = v' \\ \big(\ell(balance) < v' \quad \text{or} \quad \mathtt{fallback} \notin C\big) \end{array}}{C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{transfer}(e');\ \mathsf{S} \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid 0}$$

[METH]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C'' \quad \llbracket \overline{e'} \rrbracket_{C',v,C,\ell} = \overline{v'} \\ \mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ z};\mathsf{S}_{\mathtt{m}}\} \in C'' \end{array}}{C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{m}(\overline{e'}) \longrightarrow C(\ell \cdot \ell') \mid C''(\ell''[\overline{x} \mapsto \overline{v'}, \overline{z} \mapsto \overline{0}] \cdot \ell''') \mid C \overset{0}{\blacktriangleright} C'' : \mathsf{S}_{\mathtt{m}}}$$

[METH-FB]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C'' \quad \llbracket \overline{e'} \rrbracket_{C',v,C,\ell} = \overline{v'} \\ \mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ z};\mathsf{S}_{\mathtt{m}}\} \notin C'' \quad \mathtt{fallback} \in C'' \end{array}}{C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{m}(\overline{e'}) \xrightarrow{\checkmark} C(\ell \cdot \ell) \mid 0}$$

[METH-ERR]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C'' \quad \llbracket \overline{e'} \rrbracket_{C',v,C,\ell} = \overline{v'} \\ \mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ z};\mathsf{S}_{\mathtt{m}}\}, \mathtt{fallback} \notin C'' \end{array}}{C(\ell \cdot \ell') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{m}(\overline{e'}) \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid 0}$$

[METH-PAY]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C'' \quad \llbracket \overline{e'} \rrbracket_{C',v,C,\ell} = \overline{v'} \quad \llbracket e'' \rrbracket_{C',v,C,\ell} = v'' \\ \mathtt{m}(\overline{\mathsf{T}\ x})\ \mathtt{payable}\ \{\overline{\mathsf{T}'\ z};\mathsf{S}_{\mathtt{m}}\}, \mathtt{fallback} \in C'' \quad \ell(balance) \geqslant v'' \\ \ell_1 = \ell[balance \mapsto^{-} v''] \quad \ell_2 = \ell''[balance \mapsto^{+} v'', \overline{x} \mapsto \overline{v'}, \overline{z} \mapsto \overline{0}] \end{array}}{C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{m}.\mathtt{value}(e'')(\overline{e'}) \longrightarrow C(\ell_1 \cdot \ell') \mid C''(\ell_2 \cdot \ell''') \mid C \overset{v''}{\blacktriangleright} C'' : \mathsf{S}_{\mathtt{m}}}$$

[METH-PAY-FB]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C'' \quad \llbracket \overline{e'} \rrbracket_{C',v,C,\ell} = \overline{v'} \quad \llbracket e'' \rrbracket_{C',v,C,\ell} = v'' \\ \mathtt{m}(\overline{\mathsf{T}\ x})\ \mathtt{payable}\ \{\overline{\mathsf{T}'\ z};\mathsf{S}_{\mathtt{m}}\} \notin C'' \quad \mathtt{fallback} \in C'' \\ \ell(balance) \geqslant v'' \quad \ell_1 = \ell[balance \mapsto^{-} v''] \quad \ell_2 = \ell''[balance \mapsto^{+} v''] \end{array}}{C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{m}.\mathtt{value}(e'')(\overline{e'}) \xrightarrow{\checkmark} C(\ell_1 \cdot \ell_1) \mid C''(\ell_2 \cdot \ell_2) \mid 0}$$

[METH-PAY-ERR]
$$\frac{\begin{array}{c} \llbracket e \rrbracket_{C',v,C,\ell} = C'' \quad \llbracket \overline{e'} \rrbracket_{C',v,C,\ell} = \overline{v'} \quad \llbracket e'' \rrbracket_{C',v,C,\ell} = v'' \\ \ell(balance) < v'' \quad \text{or} \quad \big(\mathtt{m}(\overline{\mathsf{T}\ x})\ \mathtt{payable}\ \{\overline{\mathsf{T}'\ z};\mathsf{S}_{\mathtt{m}}\}, \mathtt{fallback} \notin C''\big) \end{array}}{C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C' \overset{v}{\blacktriangleright} C : e.\mathtt{m}.\mathtt{value}(e'')(\overline{e'}) \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid C''(\ell''' \cdot \ell''') \mid 0}$$

[CMT]
$$\frac{\mathcal{S} \xrightarrow{\checkmark} \mathcal{S}'}{C(\ell \cdot \ell') \mid \mathcal{S} \xrightarrow{\checkmark} C(\ell \cdot \ell) \mid \mathcal{S}'}$$

[BKT]
$$\frac{\mathcal{S} \xrightarrow{\mathsf{fail}} \mathcal{S}'}{C(\ell \cdot \ell') \mid \mathcal{S} \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid \mathcal{S}'}$$

[TAU]
$$\frac{\mathcal{S} \longrightarrow \mathcal{S}'}{C(\ell \cdot \ell') \mid \mathcal{S} \longrightarrow C(\ell \cdot \ell') \mid \mathcal{S}'}$$

Fig. 2. State transitions $\xrightarrow{\mu}$ of mSCL ($\xrightarrow{\mu} = \longrightarrow \cup \xrightarrow{\checkmark} \cup \xrightarrow{fail}$), $\llbracket e \rrbracket_{C',v,C,\ell}$ never fails.

ℓ'), we always assume that $dom(\ell') = fields(C) \subseteq dom(\ell)$ (because ℓ also defines formal parameters and local variables). We say that a state is final when the runtime statement is of the form $\prod_{i \in I} C_i(\ell_i \cdot \ell_i)|0$. Note that in a final state the two memories of every contract are equal. Contracts $C(\ell \cdot \ell')$ have a unique name C that is in one to one correspondence with contract names in $\mathcal{P}$.

Runtime statements may be either 0, the terminated statement, or $C' \overset{v}{\blacktriangleright} C$: S, where S must be evaluated into the contract C, with a caller C' and with a value v.

The semantics of mSCL programs is defined by means of the transition relation $\mathcal{S} \xrightarrow{\mu} \mathcal{S}'$, where $\xrightarrow{\mu} = \longrightarrow \cup \xrightarrow{\checkmark} \cup \xrightarrow{\mathsf{fail}}$ (the program is kept implicit in the notation). In a $\xrightarrow{\mu}$-derivation to a final state, all transitions are $\longrightarrow$, except the last one that is responsible for committing the memory. In particular, if the last transition is a $\xrightarrow{\checkmark}$, then the computation terminates normally and the current memory becomes the new initial memory; if the last transition is $\xrightarrow{\mathsf{fail}}$ then the computation backtracks and the memory is reverted to the initial memory. The formal definition of $\xrightarrow{\mu}$ is given in Fig. 2.

Let us comment on some semantic rules (comments are omitted when rules are standard). Rule [UPD] defines the semantics of an update of a field or a variable: the expression e is evaluated in the current memory of C and the resulting memory binds the value to x. Rules [TRANSFER] and [TRANSFER-FAIL] define the semantics of e.transfer(e'). The former one verifies that the recipient e is payable (e.g. has a fallback function) and the caller's balance is larger than e'; in this case, the balances of the caller and of e are updated. The second rule deals with errors: either the recipient is not payable or the caller's is not sufficient. In this case, a failure occurs and it is propagated to the whole solution (with rule [BKT]). When fallback bodies are nonempty, [TRANSFER] is more complex: see [TRANSFER-CONT] in Fig. 6).

Rules [METH*] of Fig. 2 deal with function invocations, which are particularly complex in mSCL. Rule [METH] defines successful non-payable function invocations $e.\mathtt{m}(\overline{e})$. In this case, the function dispatch is performed by using the value C'' of e and the statement to evaluate becomes the body of m (without any continuation). Rules [METH-FB] and [METH-ERR] define unsuccessful non-payable function invocations. The two rules deal with the two subcases whether the callee has a fallback function or not; in the first one, the invocation is dispatched to the fallback that has an empty body and the computation terminates successfully; in the second one, the invocation fails and the overall computation backtracks. The other three rules for function invocations, namely [METH-PAY], [METH-PAY-FB], and [METH-PAY-ERR] account for invocations of payable functions. In these cases, the invocation carries a value and, when it is successful, the balances of the caller and of the callee must be updated correspondingly. Rule [METH-PAY-ERR] does not update balances because it models a failure. This happens either when the caller's balance is smaller than the value to be sent or when the dispatch cannot be performed because there is no function and there is no fallback.

3.1. Initial states

The initial state of an mSCL program $\mathcal{P} = (\mathsf{C}_1, \cdots, \mathsf{C}_n)$ is a term

$$\prod_{i\in 1..n} C_i(\ell_i \cdot \ell_i)|\bot \overset{0}{\blacktriangleright} C' : C.\mathtt{m}(\overline{v})$$

where $\bot$ is a dummy smart contract name, $C, C' \in cnames(\mathcal{P})$ and C_i is the contract name of C_i. That is we assume that runtime contracts are in a one-to-one correspondence with smart contract definitions; we duplicate the code in case we need several runtime contracts of a same C. We also assume that *Id* contains a dummy name *User* that may be used instead of C' in the initial state. We use this expedient in order to cover invocations of a function of the program by an external smart contract or by an external user. To reduce the number of cases in the translations and analyses, we are ruling out initial statements such as $C.\mathtt{m}.\mathtt{value}(v')(\overline{v})$ from the formal parts of the paper, but we will use them nevertheless in the examples.

For example, the initial state of Example 1 is

$$Bank(\ell_B \cdot \ell_B)|Thief(\ell_T \cdot \ell_T)|\bot \blacktriangleright Thief : \mathtt{Bank.pay.value}(1)(2)$$

where $\ell_B = [balance \mapsto v]$ and $\ell_T = [balance \mapsto 1]$.

We conclude by observing that mSCL programs are executed sequentially, in a deterministic way, and that the execution never gets stuck.

Theorem 2. (***Determinism and progress***) *Let $\mathcal{P}$ be an mSCL program and $\mathcal{S}$ be an initial state such that $\mathcal{S} \longrightarrow \mathcal{S}'$. Then*

1. *Determinism: there is at most one $\mathcal{S}''$ such that $\mathcal{S}' \xrightarrow{\mu} \mathcal{S}''$.*
2. *Progress: either $\mathcal{S}' \longrightarrow \mathcal{S}''$ for some $\mathcal{S}''$, or $\mathcal{S}'$ is final.*

We note that progress is a consequence of the assumption that mSCL programs are well-typed (Assumption 1), which ensures that all invocations of $[\![e]\!]_{C',v,C,\ell}$ return a value and that when e is of type *address* then $[\![e]\!]_{C',v,C,\ell} = C''$ where C'' is the name of one of the contracts in the state.

4. The translation of mSCL into an intermediate language

Programs in our intermediate language are sets of functions that take in input two environments—the backtrack one and the current one—and variables (which represent non-negative integers and smart contract names) and return an environment. Environments, which are native values in the intermediate language, encode the state of an mSCL program, namely they map fields and local variables to values. More precisely, the codomain of environments are abstract values that are expressions of mSCL[2]. As we will see, the evaluation of a program amounts to computing a final environment from the initial ones, which are identical, by passing updated current environments from one function invocation to another.

4.1. Environments

Γ, called environment, is a map $(Id \rightarrow FId \rightarrow \mathsf{E}) \cup (Var \rightarrow \mathsf{E})$; we always shorten $\Gamma(C)(\mathtt{f})$ into $\Gamma(C.\mathtt{f})$ and use $\Gamma[C.\mathtt{f} \mapsto e]$ to denote the update to e of the field $C.\mathtt{f}$. Notice that environments return abstract terms (which are expressions in mSCL) rather than (integer) values. We also use two update operations on environments: $\Gamma[C.\mathtt{f} \mapsto^{+} e] \overset{\text{def}}{=} \Gamma[C.\mathtt{f} \mapsto \Gamma(C.\mathtt{f}) + e]$ and $\Gamma[C.\mathtt{f} \mapsto^{-} e] \overset{\text{def}}{=} \Gamma[C.\mathtt{f} \mapsto \Gamma(C.\mathtt{f}) - e]$.

The syntax of the intermediate language uses particular environments, called pure: an environment is pure whenever it is injective and returns only variables. The semantics of the intermediate language also uses ground environments: an environment is a ground when the expressions in the codomain are ground values.

4.2. Syntax of the intermediate language

A program in the intermediate language is a tuple $\mathcal{I}$ of function definitions

$$C.m(\Gamma_0, \Gamma_1, v, \overline{x}, H) = \sum_{D \in Id} (H = D)\Theta_D$$

(we keep the notation of mSCL for the name of functions). We require that

1. the formal parameters of a function definition include two environments Γ_0 and Γ_1 that are pure and with disjoint codomains. Γ_0 is the environment that has to be returned in the case of backtrack; Γ_1 is the environment that must be updated by the function body in the case of successful termination;
2. the remaining parameters, namely $v, \overline{x}$ and H, respectively, describe the amount of the transferred cryptocurrency, the parameters of the function and the caller name.

We observe that functions' bodies are summands on the set *Id* of smart contract names that, for every program, we assume to be finite. This expedient allows us to consider only ground smart contract names during the translation. This is the technique we use to map mSCL, which has dynamic address resolution, to a language with static dispatch only.

The syntax of function bodies Θ is

$$\Theta \ ::= \quad \Gamma \quad | \quad e.m(\Gamma, \Gamma', e', \overline{e''}, H) \quad | \quad \sum_{i\in 1..n} (\phi_i) \ \Theta_i$$

where φ_i are boolean expressions that also contain predicates such as $m \in C$ or $m.payable \in C$. According to the syntax, a function may either return an environment, or invoke another function, or have a nondeterministic behavior $\sum_{i\in 1..n}(\varphi_i)\ \Theta_i$ that is regulated by a finite set of predicates $\varphi_1, \cdots, \varphi_n$. The term $\sum_{i\in 1..n}(\varphi_i)\ \Theta_i$ is an abbreviation for $(\varphi_1)\ \Theta_1 + \cdots + (\varphi_n)\ \Theta_n$ (we use the latter notation when we write programs).

4.3. Semantics of the intermediate language

In order to formalize the semantics of function call, we need to match an actual parameter Γ' that is a ground environment with the formal one Γ that is a pure environment. We denote with $\sigma_{\Gamma,\Gamma'}$ the unique substitution such that $\sigma_{\Gamma,\Gamma'} \circ \Gamma = \Gamma'$.

[2] To improve readability, we denote with e the expressions that occur in mSCL programs and with e the same expressions when used as abstract values in the intermediate language. Equivalently, e and e range over the productions of two grammars E and E that are defined identically.

The semantics of a program is defined by the two rules:

$$\texttt{[Apply]}\quad \frac{\left(C.m(\Gamma_0,\Gamma_1,x,\overline{z},H) = \sum_{i\in 1..n}(H = D_i)\Theta_{D_i}\right) \in \mathcal{I} \qquad 1 \le k \le n \qquad [\![e]\!] = u \qquad [\![\overline{e'}]\!] = \overline{v}}{C.m(\Gamma,\Gamma',e,\overline{e'},D_k) \Rightarrow_{\mathcal{I}} \Theta_{D_k}\{u,\overline{v}/x,\overline{z}\}\sigma_{\Gamma_0,\Gamma}\sigma_{\Gamma_1,\Gamma'}}$$

$$\texttt{[Choice]}\quad \frac{[\![\phi_i]\!] = true}{\sum_{i\in I}(\phi_i)\Theta_i \Rightarrow_{\mathcal{I}} \Theta_i}$$

where [[*e*]] is the value of *e*. (The definition of [[*e*]] is omitted because straightforward.) We notice that the semantics of intermediate programs is nondeterministic: if Θ is $(1>0)\Theta_1 + (2>1)\Theta_2$ then it may evolve into either Θ_1 or Θ_2. We also notice that the intermediate language is actually a standard functional language with mappings (the environments), tuples, conditionals, and nondeterminism. Rule [Apply] is beta-reduction plus pattern matching over mappings, while rule [Choice] allows one to select a branch when the corresponding guard is true. The syntax and the semantics of the intermediate language are illustrated in the following example.

Example 3. The function `Bank.pay` and `Thief.ack` of Example 1 can be written in the intermediate language as follows. Let

$$\begin{array}{lcl}\Gamma_0 & = & \left[Bank \mapsto \left[balance \mapsto x_{Bank,b}\right], Thief \mapsto \left[balance \mapsto x_{Thief,b}\right]\ \right] \\ \Gamma_1 & = & \left[Bank \mapsto \left[balance \mapsto y_{Bank,b}\right], Thief \mapsto \left[balance \mapsto y_{Thief,b}\right]\ \right]\end{array}$$

Notice that Γ_0 and Γ_1 are pure environments with disjoint codomains. Let also *Id* = {Bank, Thief}. For `Bank.pay` we obtain:

$$\begin{array}{l} Bank.pay(\Gamma_0,\Gamma_1,v,n,H) = \sum_{D\in Id}(H=D) \quad \left(v \ge 1 \wedge y_{Bank,b} > n \wedge n < 5\right)\Theta \\ \qquad\qquad\qquad\qquad + \ !\left(v \ge 1 \wedge y_{Bank,b} > n \wedge n < 5\right)\Gamma_1 \end{array}$$

$$\begin{array}{rcl} \text{where}\quad \Theta & = & \left(y_{Bank,b} > n \wedge fallback \in D\right)\Theta' + \left(y_{Bank,b} \le n\right)\Gamma_0 + (fallback \notin D)\Gamma_0 \\ \Theta' & = & (ack \in D)\, D.ack\left(\Gamma_0,\Gamma'_1,0,Bank\right) \\ & & +\ (ack.payable \in D)\, D.ack\left(\Gamma_0,\Gamma'_1,0,Bank\right) \\ & & +\ (ack \notin D \wedge ack.payable \notin D \wedge fallback \in D)\,\Gamma'_1 \\ & & +\ (ack \notin D \wedge ack.payable \notin D \wedge fallback \notin D)\,\Gamma_0 \\ \Gamma'_1 & = & \Gamma_1[Bank.balance \mapsto^{-} n, D.balance \mapsto^{+} n] \end{array}$$

For `Thief.ack` we get:

$$Thief.ack(\Gamma_0,\Gamma_1,v,H) = \sum_{D\in Id}(H=D)\,\Theta''$$

$$\begin{array}{rcl} \text{where}\quad \Theta'' & = & (pay \in D)\,\Gamma_0 \\ & & +\ \left(pay.payable \in D \wedge y_{Thief,b} \ge 1\right)\Theta'' \\ & & +\ \left(pay.payable \in D \wedge y_{Thief,b} < 1\right)\Gamma_0 \\ & & +\ \left(pay \notin D \wedge pay.payable \notin D \wedge fallback \in D \wedge y_{Thief,b} \ge 1\right)\Gamma'_1 \\ & & +\ \left(pay \notin D \wedge pay.payable \notin D \wedge fallback \in D \wedge y_{Thief,b} < 1\right)\Gamma_0 \\ & & +\ (pay \notin D \wedge pay.payable \notin D \wedge fallback \notin D)\,\Gamma_0 \end{array}$$

and $\Theta'' = D.pay\left(\Gamma_0,\Gamma'_1,1,2,Thief\right)$ and $\Gamma'_1 = \Gamma_1[Thief.balance \mapsto^{-} 1, D.balance \mapsto^{+} 1]$

As regards the semantics of the intermediate language, let us discuss the transitions of *Bank.pay*(*Γ*, *Γ*, 1, 2, *Thief*), where $\Gamma = [Bank \mapsto [balance \mapsto 4], Thief \mapsto [balance \mapsto 1]\ ]$.

$$\begin{array}{rl} Bank.pay(\Gamma,\Gamma,1,2,Thief) \Rightarrow_{\mathcal{I}} & (1 \ge 1 \wedge 4 > 1 \wedge 2 < 5)\,\Theta\{Thief/D\}\{2,4,1/n,y_{Bank,b},y_{Thief,b}\} \\ & +!(1 \ge 1 \wedge 4 > n \wedge 2 < 5)\,\Gamma \\ \Rightarrow_{\mathcal{I}} & (ack \in Thief)\, Thief.ack(\Gamma,\Gamma',0,Bank) \\ & +\ (ack.payable \in Thief)\, Thief.ack(\Gamma,\Gamma',0,Bank) \\ & +\ (ack \notin Thief \wedge ack.payable \notin Thief \wedge fallback \in Thief)\,\Gamma' \\ & +\ (ack \notin Thief \wedge ack.payable \notin Thief \wedge fallback \notin Thief)\,\Gamma \\ \Rightarrow_{\mathcal{I}} & Thief.ack(\Gamma,\Gamma',0,Bank) \\ \Rightarrow_{\mathcal{I}} & \Theta''\{Bank/D\}\{0,2,3/v,y_{Bank,b},y_{Thief,b}\} \\ \Rightarrow_{\mathcal{I}} & Bank.pay(\Gamma,\Gamma'',1,2,Thief) \end{array}$$

where

$$\Gamma' = [Bank \mapsto [balance \mapsto 2], Thief \mapsto [balance \mapsto 3]\ ]$$

$$\Gamma'' = [Bank \mapsto [balance \mapsto 3], Thief \mapsto [balance \mapsto 2]\ ]\ .$$

4.4. The translation of `mSCL`

The translation of `mSCL` in the intermediate language is defined by using judgments and inference rules. The judgments have the following form:

- *judgments for expressions*: $\Gamma \vdash^{e}_{C,D} \mathsf{E} : e'$, where *e* and *e′* are expressions that contain constants or variables; *e* is the amount of cryptocurrency transmitted during the invocation, while *e′* is the value of the expression E; *C* and *D* are respectively the caller and the callee contracts;
- *judgments for statements*: $\Gamma, \Gamma' \vdash^{e}_{C,D} \mathsf{S} : \Theta$, where Γ is the backtrack environment, Γ′ is the current environment and Θ is the resulting intermediate code (*e*, *C* and *D* are similar to the corresponding one for judgments of expressions). Backtrack and current environments correspond to the (instances of) environments Γ_0 and Γ_1 in the function definitions and are used to model backtrack (in case of failures) and success, respectively.

The translation of expressions is reported in Fig. 3. It partially evaluates expressions by replacing accesses to fields with the corresponding values in the environment. Rules [field] and [var] manage variables; there are three cases: a variable is a callee's field, or it is a formal parameter or a smart contract name. In any case, we return the corresponding value in Γ (which may also be an expression). In [bal] the translation of *e′.balance* is the balance of a contract; in this case, it is necessary that *e′* is a smart contract name *H*: in our setting we write *H* ∈*Id*. Rule [prod-div] addresses multiplication and division. Since the cost analysis in Section 5 only covers Presburger arithmetics expressions where the second argument of products and divisions are constants, the inference rules do not translate expressions that cannot be fed to the analyzer. The translation of statements is defined in Fig. 4. The judgments return intermediate codes that use the predicates (the notation is the same that has been used in `mSCL`):

$$\textsc{[const]}\ \frac{v \in \texttt{uint} \quad \text{or} \quad v \in Id}{\Gamma \vdash^{e}_{C,D} v : v} \qquad \textsc{[field]}\ \frac{x \in dom(\Gamma(D))}{\Gamma \vdash^{e}_{C,D} x : \Gamma(D.x)} \qquad \textsc{[var]}\ \frac{x \notin dom(\Gamma(D))}{\Gamma \vdash^{e}_{C,D} x : \Gamma(x)} \qquad \textsc{[this]}\ \Gamma \vdash^{e}_{C,D} \texttt{this} : D$$

$$\textsc{[sender]}\ \Gamma \vdash^{e}_{C,D} \texttt{msg.sender} : C \qquad \textsc{[this]}\ \Gamma \vdash^{e}_{C,D} \texttt{msg.value} : e \qquad \textsc{[bal]}\ \frac{\Gamma \vdash^{e}_{C,D} e' : H \quad H \in Id}{\Gamma \vdash^{e}_{C,D} e'.balance : \Gamma(H.balance)}$$

$$\textsc{[ops]}\ \frac{\Gamma \vdash^{e}_{C,D} e_1 : e'_1 \quad \Gamma \vdash^{e}_{C,D} e_2 : e'_2 \quad \sharp \in \{+.-, >, =, \geqslant, \&\&\}}{\Gamma \vdash^{e}_{C,D} e_1 \sharp e_2 : e'_1 \sharp e'_2} \qquad \textsc{[prod-div]}\ \frac{\Gamma \vdash^{e}_{C,D} e' : e'' \quad \sharp \in \{*, /\}}{\Gamma \vdash^{e}_{C,D} e' \sharp n : e'' \sharp n} \qquad \textsc{[not]}\ \frac{\Gamma \vdash^{e}_{C,D} e' : e''}{\Gamma \vdash^{e}_{C,D} !e' : !e''}$$

Fig. 3. Translation of `mSCL` expressions.

[EMPTY]
$$\Gamma, \Gamma' \vdash^{e}_{C,D} \varepsilon : \Gamma'$$

[REVERT]
$$\Gamma, \Gamma' \vdash^{e}_{C,D} \mathtt{revert;} : \Gamma$$

[ASGN]
$$\frac{\begin{array}{c} x \in dom(\Gamma(D))\backslash\{balance\} \\ \Gamma' \vdash^{e}_{C,D} \mathsf{E} : e' \quad \Gamma, \Gamma'[D.x \mapsto e'] \vdash^{e}_{C,D} \mathsf{S} : \Theta \end{array}}{\Gamma, \Gamma' \vdash^{e}_{C,D} x{=}\mathsf{E};\mathsf{S} : \Theta}$$

[ASGN-VAR]
$$\frac{\begin{array}{c} x \notin dom(\Gamma(D)) \\ \Gamma' \vdash^{e}_{C,D} \mathsf{E} : e' \quad \Gamma, \Gamma'[x \mapsto e'] \vdash^{e}_{C,D} \mathsf{S} : \Theta \end{array}}{\Gamma, \Gamma' \vdash^{e}_{C,D} x{=}\mathsf{E};\mathsf{S} : \Theta}$$

[INVK-NV]
$$\frac{\Gamma' \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in Id \quad \Gamma' \vdash^{e}_{C,D} \overline{\mathsf{E}} : \overline{e'}}{\Gamma, \Gamma' \vdash^{e}_{C,D} \mathsf{E}.\mathtt{m}(\overline{\mathsf{E}}) : \begin{array}{ll} & (m{\in}e_0)\ e_0.m(\Gamma,\Gamma',0,\overline{e'},D) \\ + & (m.payable{\in}e_0)\ e_0.m(\Gamma,\Gamma',0,\overline{e'},D) \\ + & (m{\notin}e_0 \wedge m.payable{\notin}e_0 \wedge fallback{\in}e_0)\ \Gamma' \\ + & (m{\notin}e_0 \wedge m.payable{\notin}e_0 \wedge fallback{\notin}e_0)\ \Gamma \end{array}}$$

[INVK]
$$\frac{\begin{array}{c} \Gamma' \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in Id \quad \Gamma' \vdash^{e}_{C,D} \overline{\mathsf{E}} : \overline{e'} \quad \Gamma' \vdash^{e}_{C,D} \mathsf{E}' : e'' \\ \Gamma'' = \Gamma'[e_0.balance \mapsto^{+} e''][D.balance \mapsto^{-} e''] \end{array}}{\Gamma, \Gamma' \vdash^{e}_{C,D} \mathsf{E}.\mathtt{m}.\mathtt{value}(\mathsf{E}')(\overline{\mathsf{E}}) : \begin{array}{ll} & (m{\in}e_0)\ \Gamma \\ + & (m.payable{\in}e_0 \wedge \Gamma'(D.balance){\geqslant}e'')\ e_0.m(\Gamma,\Gamma'',e'',\overline{e'},D) \\ + & (m.payable{\in}e_0 \wedge \Gamma'(D.balance){<}e'')\ \Gamma \\ + & (m{\notin}e_0 \wedge m.payable{\notin}e_0 \wedge fallback{\in}e_0 \wedge \Gamma'(D.balance){\geqslant}e'')\ \Gamma'' \\ + & (m{\notin}e_0 \wedge m.payable{\notin}e_0 \wedge fallback{\in}e_0 \wedge \Gamma'(D.balance){<}e'')\ \Gamma \\ + & (m{\notin}e_0 \wedge m.payable{\notin}e_0 \wedge fallback{\notin}e_0)\ \Gamma \end{array}}$$

[IF-THEN-ELSE]
$$\frac{\Gamma' \vdash^{e}_{C,D} \mathsf{E} : e' \quad \Gamma, \Gamma' \vdash^{e}_{C,D} \mathsf{S} : \Theta \quad \Gamma, \Gamma' \vdash^{e}_{C,D} \mathsf{S}' : \Theta'}{\Gamma, \Gamma' \vdash^{e}_{C,D} \mathtt{if}\ (\mathsf{E})\ \{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}'\ \} : (e')\ \Theta + (!e')\ \Theta'}$$

[TRANSFER]
$$\frac{\begin{array}{c} \Gamma' \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in Id \quad \Gamma' \vdash^{e}_{C,D} \mathsf{E}' : e' \\ \Gamma'' = \Gamma'[e_0.balance \mapsto^{+} e'][D.balance \mapsto^{-} e'] \\ \Gamma, \Gamma'' \vdash^{e}_{C,D} \mathsf{S} : \Theta \end{array}}{\Gamma, \Gamma' \vdash^{e}_{C,D} \mathsf{E}.\mathtt{transfer}(\mathsf{E}');\mathsf{S} : \begin{array}{ll} & (\Gamma'(D.balance){\geqslant}e' \wedge fallback{\in}e_0)\ \Theta \\ + & (\Gamma'(D.balance){<}e')\ \Gamma \\ + & (fallback{\notin}e_0)\ \Gamma \end{array}}$$

Fig. 4. The translation of `mSCL` statements.

- *fallback* $\in e$, with $e \in Id$, to mean that the contract e has the fallback function;
- $m \in e$, with $e \in Id$, to mean that m is a function in e that is not payable; *m.payable* $\in e$ additionally requires that m is also payable.

The translation of statements is defined in Fig. 4. Rules [INVK-NV] and [INVK] define function invocations for non-payable functions and payable ones, respectively. The former one returns a choice between several alternatives: (i) when m is in e_0 then it reduces to the invocation; (ii) when m is in e_0 and it is payable then it is translated to the invocation with 0 cryptocurrency transferred; (iii) when m is not in e_0 but the contract has the fallback function then the translation is the call to fallback that, in our case, returns the current environment (because fallback has an empty body); (iv) when both m and fallback are not in e_0 then a backtrack occurs and the translation is the backtrack environment. Rule [INVK] manages invocations with cryptocurrency transfer from the caller to the callee; in this case, we must check that the caller has enough cryptocurrency in his balance, otherwise a backtrack occurs.

The translation of `mSCL` is completed with the rules for function definition and programs, given in Fig. 5, where we use the judgments $\Gamma_1 \vdash C.m(\Gamma_0, \Gamma_1, v, \overline{x}, H) = \Theta$ and $\vdash \mathcal{P} : \mathcal{I}$ with the obvious meaning. In [FUNCTION], the definition of a function is given in two pure environments that act as formal parameters. We recall that Γ_0 is the backtrack environment, e.g. the environment to which transiting in case of errors, while Γ_1 is the environment where the function invocation must be evaluated.

The critical point is that, in our system, the set *Id* is finite, therefore the hypotheses of rule [FUNCTION] and the choice in the conclusion are finite. (Said otherwise, we analyze the cost of smart contract programs with a finite number of known contract instances.) Rule [PROGRAM] gives the translation of a smart contract program. The premise of the rule contains a set of hypotheses that depend on a finite set of smart contract names and function names. This does not mean that our analysis requires that the code of all the interacting contracts must be known. In particular, the analysis (and our prototype) covers invocations of a function of the program by an external caller (either a smart contract or a user). As discussed in Section 3, we assume the presence of a dummy name *User* that belongs to *Id*.

As an example, one can compute the translation defined in this section when applied to the corresponding functions of the `mSCL` program in Fig. 1. The reader may verify that these codes are exactly those of *Bank.pay* and *Thief.ack* in Example 3.

We conclude this section by asserting the correctness of the translation. To assess this property we need to formalize the correspondence between a state of an `mSCL` program and its intermediate code. The following definition intends to specify this relationship.

Definition 4. (Correspondence of states and intermediate codes) Given a state $\mathcal{S} = \prod_{i\in 1..n} C_i(\ell'_i \cdot \ell_i) | C_k \overset{v}{\blacktriangleright} C_h : \mathsf{S}$, we define

$$envs(\mathcal{S}) \overset{\mathsf{def}}{=} \left[(C_i \mapsto \ell_i)^{i\in 1..n}\right],\ \left[(C_i \mapsto \ell'_i)^{i\in(1..n)\backslash h}, C_h \mapsto \ell'_h|_{fields(C_h)}, \ell'_h|_{Var}\right]$$

[FUNCTION]
$$\frac{\begin{array}{c} \left(\begin{array}{l} \Gamma_0(D') = [\mathtt{f}_1 \mapsto x_{D',1}, \cdots, \mathtt{f}_n \mapsto x_{D',n}, balance \mapsto x_{D',b}] \\ \Gamma_1(D') = [\mathtt{f}_1 \mapsto y_{D',1}, \cdots, \mathtt{f}_n \mapsto y_{D',n}, balance \mapsto y_{D',b}] \end{array}\right)^{\{\mathtt{f}_1,\cdots,\mathtt{f}_n, balance\} = fields(D'), D' \in Id} \\ \mathtt{function}\ \mathtt{m}(\overline{\mathsf{T}\ x})[\mathtt{payable}]\{\overline{\mathsf{T}\ y};\ \mathsf{S}\} \in C \qquad \left(\Gamma_0, \Gamma_1[\overline{x} \mapsto \overline{x_0}, \overline{y} \mapsto \overline{0}] \vdash^{v}_{D,C} \mathsf{S} : \Theta_D\right)^{D\in Id} \end{array}}{\Gamma_0, \Gamma_1 \vdash C.m(\Gamma_0,\Gamma_1,v,\overline{x_0},H) = \sum_{D\in Id}(H{=}D)\ \Theta_D}$$

[PROGRAM]
$$\frac{\mathcal{I} = \left(\Gamma_0, \Gamma_1 \vdash C.m(\Gamma_0,\Gamma_1,v,\overline{x},H) = \Theta_{C.m}\right)^{C\in cnames(\mathcal{P})}}{\vdash \mathcal{P} : \mathcal{I}}$$

Fig. 5. The translation for `mSCL` functions and programs.

[EMPTY-CONT]

$$\frac{\left(\begin{array}{c}\Gamma_1' = \Gamma_1[C'.\mathit{balance} \mapsto^{+} e', D'.\mathit{balance} \mapsto^{-} e'] \\ \Gamma_0, \Gamma_1' \vdash^{e,\sigma\perp\!\!\!\perp}_{C,C'} D'.\mathtt{m}.\mathtt{value}(e')(\overline{e''}\downarrow_{D'.\mathtt{m}}) : \Theta_{C',D',\mathtt{m}}\end{array}\right)^{C',D'\in Id,\mathtt{m}\in \mathit{fun}(D')}}{\Gamma_0,\Gamma_1 \vdash^{e,\langle H,m,\overline{e''},H',e'\rangle\sigma}_{C,D} \varepsilon : \begin{array}{l}\sum_{C',D'\in Id, m\in \mathit{fun}(D')}(H{=}D' \wedge H'{=}C')\Theta_{C',D',m} \\ + \quad (H{=}\perp)\ \Gamma_1\end{array}}$$

[REVERT-CONT]

$$\Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} \mathtt{revert;} : \Gamma_0$$

[ASGN-CONT]

$$\frac{\begin{array}{c} x\in \mathit{fields}(\Gamma_0(D))\setminus\{\mathit{balance}\} \\ \Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e' \quad \Gamma_0, \Gamma_1[D.x\mapsto e'] \vdash^{e,\sigma}_{C,D} \mathsf{S} : \Theta\end{array}}{\Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} x\mathtt{=}\mathsf{E};\mathsf{S} : \Theta}$$

[ASGN-VAR-CONT]

$$\frac{\begin{array}{c} x\notin \mathit{fields}(\Gamma_0(D)) \\ \Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e' \quad \Gamma_0, \Gamma_1[x\mapsto e'] \vdash^{e,\sigma}_{C,D} \mathsf{S} : \Theta\end{array}}{\Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} x\mathtt{=}\mathsf{E};\mathsf{S} : \Theta}$$

[IF-THEN-ELSE-CONT]

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e' \\ \Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} \mathsf{S}\ ;\ \mathsf{S}'' : \Theta \quad \Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} \mathsf{S}'\ ;\ \mathsf{S}'' : \Theta'\end{array}}{\Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} \mathtt{if}\ (\mathsf{E})\ \{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}'\ \}\ ;\ \mathsf{S}'' : (e')\ \Theta + (!e')\ \Theta'}$$

[TRANSFER-CONT]

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0\in Id \quad \Gamma_1 \vdash^{e}_{C,D} \mathsf{E}' : e' \\ \Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} e_0.\mathtt{fallback}.\mathtt{value}(e')(\ )\ ;\ \mathsf{S} : \Theta\end{array}}{\Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} \mathsf{E}.\mathtt{transfer}(\mathsf{E}');\mathsf{S} : \Theta}$$

[RETURN-CONT]

$$\frac{\Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e_{ret} \qquad \left(\begin{array}{c}\Gamma_1' = \Gamma_1[C'.\mathit{balance} \mapsto^{+} e', D'.\mathit{balance} \mapsto^{-} e'] \\ \Gamma_0, \Gamma_1' \vdash^{e,\sigma\perp\!\!\!\perp}_{C,C'} D'.\mathtt{m}.\mathtt{value}(e').(e_{ret},\overline{e''}\downarrow_{D'.\mathtt{m}-1}) : \Theta_{C',D',\mathtt{m}}\end{array}\right)^{C',D'\in Id,\mathtt{m}\in \mathit{fun}(D')}}{\Gamma_0,\Gamma_1 \vdash^{e,\langle H,\mathtt{m},\overline{e''},H',e'\rangle\sigma}_{C,D} \mathtt{return}(\mathsf{E}) : \begin{array}{l}\sum_{C',D'\in Id, \mathtt{m}\in \mathit{fun}(D')}(H{=}D' \wedge H'{=}C')\Theta_{C',D',\mathtt{m}} \\ + \quad (H{=}\perp)\ \Gamma\end{array}}$$

[RETURN-INVK-CONT]

$$\frac{\Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} \mathsf{E}.\mathtt{m}.\mathtt{value}(\mathsf{E}')(\overline{\mathsf{E}''}) : \Theta}{\Gamma_0,\Gamma_1 \vdash^{e,\sigma}_{C,D} \mathtt{return}(\mathsf{E}.\mathtt{m}.\mathtt{value}(\mathsf{E}')(\overline{\mathsf{E}''})) : \Theta}$$

Fig. 6. Translation of mSCL statements with continuations, Part I.

and we write $\mathcal{S}\vdash\Theta$ whenever $\mathcal{S}=\prod_{i\in 1..n}C_i(\ell_i'\cdot\ell_i)|C \overset{v}{\blacktriangleright} D : \mathsf{S}$ and $\mathit{envs}(\mathcal{S})\vdash^{v}_{C,D}\mathsf{S} : \Theta$ or $\mathcal{S}=\prod_{i\in 1..n}C_i(\ell_i'\cdot\ell_i)|0$ and $\Theta=\Gamma$ and $\mathit{envs}(\mathcal{S}) = \Gamma,\Gamma$.

The correctness of the translation follows; the proof can be found in the Appendix.

Theorem 5. *Let $\mathcal{P}$ be an* mSCL *program such that $\vdash\mathcal{P}:\mathcal{I}$ and let $\mathcal{S}$ be an initial state such that $\mathcal{S}\vdash\Theta$. Then*

1. *(determinism) If $\Theta\Rightarrow_{\mathcal{I}}^{*}\Theta'$ then there is at most one Θ'' such that $\Theta'\Rightarrow_{\mathcal{I}}\Theta''$;*
2. *(correctness) If $\mathcal{S}\longrightarrow\mathcal{S}'$ then there is a Θ' such that $\mathcal{S}'\vdash\Theta'$ and $\Theta\Rightarrow_{\mathcal{I}}^{*}\Theta'$.*

5. The analysis of smart contract balances

5.1. The cost model of mSCL

The programs in the intermediate language that are generated by the translation in Section 4 return environments when they terminate. This output is too informative since we are interested in computing cryptocurrency movements of exactly one smart contract, which are recorded in the corresponding balance field. Moreover, instead of computing the final value of the balance, it is more relevant to compute an upper bound of the amount of cryptocurrencies that a smart contract can either *lose* or *gain* during a terminating computation. It is also worth noticing that the upper bounds we are looking for are not just numbers, i.e. a maximal value that can be reached considering all possible outputs. Instead, we are interested in symbolic upper bounds expressed as functions on the value of the fields of the initial environments and the actual parameters of the initial call.

We start by defining the final gain and loss associated to a smart contract C', a system of equations $\mathcal{I}$ and an initial invocation $C.m(\Gamma,\Gamma,e,\overline{x},H)$.

Definition 6. Let $\mathcal{P}$ be an mSCL program and $\vdash\mathcal{P}:\mathcal{I}$. Let $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$ and $\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}$ be the functions

$$\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}(\Gamma,z,\overline{x},H)=\begin{cases} max(0,\Gamma'(C'.balance)-\Gamma(C'.balance)) & \text{if } C.m(\Gamma,\Gamma,z,\overline{x},H)\Rightarrow^{*}_{\mathcal{I}}\Gamma' \\ 0 & \text{otherwise}\end{cases}$$

$$\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}(\Gamma,z,\overline{x},H)=\begin{cases} max(0,\Gamma(C'.balance)-\Gamma'(C'.balance)) & \text{if } C.m(\Gamma,\Gamma,z,\overline{x},H)\Rightarrow^{*}_{\mathcal{I}}\Gamma' \\ 0 & \text{otherwise}\end{cases}$$

where, for every $D\in \mathit{cnames}(\mathcal{P})$, $\mathit{dom}(\Gamma(D)) = \mathit{fields}(\mathcal{D})$. By Theorem 5(1) the above functions are well defined on ground inputs because the reduction of compiled mSCL programs is deterministic.

We notice that $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$ and $\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}$ compute the amount of cryptocurrency gained/lost at the end of the computation of $C.m(\Gamma,\Gamma,z,\overline{x},H)$. In this respect, the two functions return 0 if the computation does not terminate. Indeed, in actual smart contract languages, a divergent program will be considered to gain/lose no cryptocurrency, since the transaction will be rolled-back because either it fails or it runs out of gas—gas exhaustion turns diverging computations into failing ones. (A similar remark might concern computations that become stuck, but this never happens in our case.) We also notice that the function $\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}$ is not the opposite of $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$. For example, if C begins with a balance 10 and terminates with a balance 5, then its gain is 0 and its loss is 5.

Assuming a pointwise ordering between functions, we are interested in possible precise *upper bounds* of $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$ and $\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}$. However, sometimes our technique returns asymptotic upper bounds that are less informative, like in Example 15.

Definition 7.

- A function $\mathrm{UGAIN}^{C'}_{\mathcal{I},C,m}$ is an upper bound of $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$ if and only if, for every $\Gamma,v',\overline{v},D$ in the domain of definition of $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$, $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}(\Gamma,v',\overline{v},D)\le \mathrm{UGAIN}^{C'}_{\mathcal{I},C,m}(\Gamma,v',\overline{v},D)$.

- A function $\mathrm{UGAIN}^{C'}_{\mathcal{I},C,m}$ is an asymptotic upper bound of $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$ if and only if $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m} \in \mathcal{O}(\mathrm{UGAIN}^{C'}_{\mathcal{I},C,m})$.
- Similarly for $\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}$.

Definitions 6 and 7 are given in the intermediate language. Similar definitions may be given for `mSCL` where, this time, the input of the function is an initial state. Once they are in place, it is possible to demonstrate their relationship as a corollary of Theorem 5.

Definition 8. Let C be an `mSCL` program. Let $\mathtt{m}\mathrm{GAIN}^{C}_{\mathcal{P}}$ and $\mathtt{m}\mathrm{LOSS}^{C}_{\mathcal{P}}$ be the functions defined on initial states $\mathcal{S}$:

$$\mathtt{m}\mathrm{GAIN}^{C}_{\mathcal{P}}(\mathcal{S}) = \begin{cases} \mathtt{max}(0, C.balance(\mathcal{S}') - C.balance(\mathcal{S})), \\ \qquad \text{if } \mathcal{S} \longrightarrow^{*} \mathcal{S}' \text{ for some } \mathcal{S}' \text{ final;} \\ 0 \qquad \text{otherwise} \end{cases}$$

$$\mathtt{m}\mathrm{LOSS}^{C}_{\mathcal{P}}(\mathcal{S}) = \begin{cases} \mathtt{max}(0, C.balance(\mathcal{S}) - C.balance(\mathcal{S}')), \\ \qquad \text{if } \mathcal{S} \longrightarrow^{*} \mathcal{S}' \text{ for some } \mathcal{S}' \text{ final;} \\ 0 \qquad \text{otherwise} \end{cases}$$

where $C.balance(\mathcal{S})$ is the value of the balance field of the smart contract C in the state $\mathcal{S}$.

Corollary 1. **(of Theorem 5)** Let $\mathcal{S} = \prod_{i\in 1..n} C_i(\ell_i \cdot \ell_i)|\bot \overset{0}{\blacktriangleright} D : C.m(\overline{v})$ be an initial state of an `mSCL` program $\mathcal{P}$ and $\vdash \mathcal{P} : \mathcal{I}$ and $\mathcal{S} \vdash C.m(\Gamma,\Gamma,0,\overline{x}, D)$, where $(\Gamma,\Gamma) = envs(\mathcal{S})$.

Then $\mathtt{m}\mathrm{GAIN}^{C'}_{\mathcal{P}}(\mathcal{S}) = \mathrm{GAIN}^{C'}_{\mathcal{I},C,m}(\Gamma, 0, \overline{x}, D)$ and $\mathtt{m}\mathrm{LOSS}^{C'}_{\mathcal{P}}(\mathcal{S}) = \mathrm{LOSS}^{C'}_{\mathcal{I},C,m}(\Gamma, 0, \overline{x}, D)$.

As a consequence of Corollary 1, instead of computing upper bounds of $\mathtt{m}\mathrm{GAIN}^{C}_{\mathcal{P}}(\mathcal{S})$ and $\mathtt{m}\mathrm{LOSS}^{C'}_{\mathcal{P}}(\mathcal{S})$, it is sufficient to do the same for programs written in our intermediate language. In turn, the intermediate code may be used as input of an additional translation that returns cost equations to be fed to a cost analyzer such as `CoFloCo` [9] and `PUBS` [10]. This will allow us to compute $\mathtt{m}\mathrm{GAIN}^{C}_{\mathcal{P}}\mathcal{S}$ and $\mathtt{m}\mathrm{LOSS}^{C'}_{\mathcal{P}}\mathcal{S}$ automatically, without any effort.

In the following, we introduce the syntax of `CoFloCo` and define the set of `CoFloCo` cost equations associated with a program in our intermediate language such that the cost model considered by `CoFloCo` is either that of $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$ or that of $\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}$.

5.2. The syntax and semantics of `CoFloCo`

Cost equation solvers take a list of equations in input that are terms [9].

$$m(\overline{x}) \;=\; e + \sum_{i\in 0..n} m_i(\overline{e_i}) \qquad [\,\varphi\,]$$

where variables occurring in the right-hand side and in φ are a subset of $\overline{x}$ and[3]

- m is a (cost) function symbol,
- e (i.e. the cost of the step) and $\overline{e_i}$ are Presburger arithmetic expressions, namely (q is a positive rational number)

$$e \;::=\; x \,|\, q \,|\, e+e \,|\, e-e \,|\, q{*}e \,|\, max(e_1, \cdots, e_k)$$

- φ is a conjunction of linear constraints, e.g. constraints of the form $\ell_1 < \ell_2$ or $\ell_1 \le \ell_2$ or $\ell_1 = \ell_2$, where both ℓ_1 and ℓ_2 are Presburger arithmetic expressions.

The solution of a cost program is the computation of bounds of a particular function symbol (typically one of the first equations in the list). The bounds are parametric in the formal parameters of the function symbol. The operational semantics of the (subset of) `CoFloCo` we are considering are defined below.

Definition 9. **(Semantics of cost equations seen as a functional language)** Let $\rightarrow_{\mathrm{CoFloCo}}$ be the reduction relation over ground Presburger expressions augmented with function calls (in the obvious way) defined by the following two rewriting rules, that can be applied in any context:

1. $\mathtt{m}(\overline{e}) \rightarrow_{\mathtt{CoFloCo}} e^j\{\overline{e}/\overline{x}\} + \sum_{i=0,\ldots,n^j} \mathtt{m}^j_i(\overline{e^j_i}\{\overline{e}/\overline{x}\})$ for every cost equation

$$m(\overline{x}) \;=\; e^j + \sum_{i\in 0..n^j} m^j_i(\overline{e^j_i}) \qquad [\,\varphi^j\,]$$

such that $\varphi^j\{\overline{e}/\overline{x}\}$ holds;

2. $e \rightarrow_{\mathtt{CoFloCo}} v$ if e is a Presburger expression whose value is v.
   The relation $\rightarrow_{\mathtt{CoFloCo}}$, seen as a reduction relation, is obviously non-deterministic, as the following example shows. However, all cost equations generated from `mSCL` programs exhibit deterministic behavior.

Example 10. Consider the following set of cost equations:

$$\begin{array}{ll} n(x) = x+1 & [] \\ m(x) = 1 + n(2{*}x) & [0 \le x] \\ m(x) = 2 - n(2{*}x) & [x \le 2] \end{array}$$

It turns out that $m(1) \rightarrow_{\mathtt{CoFloCo}}{}^{*} 4$ and $m(1) \rightarrow_{\mathtt{CoFloCo}}{}^{*} -1$.

5.3. The translation

In this paragraph, we associate two sets of cost equations to every intermediate program; the first set is used to compute the upper bound for the gain of cryptocurrency of a chosen contract, while the second set is for the upper bound for the loss of cryptocurrency. The two sets of equations will only differ by the choice of a cost function that will be defined below.

Translating the codes obtained from Figs. 3–5 into cost equations does not seem difficult:

- a function in the intermediate program is mapped into a cost equation function that either returns a final environment, or it is a finite sum of function calls;
- sums are mapped to sets of guarded equations; a function call to cost equations call where the steps have 0 cost;
- returning a final environment Γ' amounts to compute the empty set of calls where the step has cost $\mathtt{max}(0, \Gamma'(C'.balance) - \Gamma(C'.balance))$ — to compute $\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}$ — or $\mathtt{max}(0, \Gamma(C'.balance) - \Gamma'(C'.balance))$ — to compute $\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}$.

In practice, the association is technically more involved due to the following differences between our intermediate language and `CoFloCo` cost equations:

- functions in our intermediate language pass around environments, while cost equations take in input tuples of variables. We will introduce a flattening operation to map the former into the latter;

[3] Actually, `CoFloCo` does not require the condition we just imposed on the variables that occur in the right-hand side. The remaining variables are handled in logic programming style, via unification. Thanks to our additional constraint, it becomes possible to think of `CoFloCo` equations like functional programs instead. We will take advantage of this later, when we will equip the syntax with an operational semantics in functional style.

- `CoFloCo` guards are very basic: only conjunctions of comparisons between integer numbers are admitted, while guards of our intermediate language use all logical operators and tests like $\mathtt{m} \in D$ that look for an element in a finite set. We will encode our expressions into `CoFloCo` guards, which will also include the writing of guards into disjunctive normal forms to fit the restricted syntax of `CoFloCo`;
- our intermediate language uses non-negative integers while `CoFloCo` uses signed integers; therefore we must be careful when encoding subtraction ($2-4=0$ on signed integers) and we must add initial preconditions to cost equations stating non-negativity of every input.

Therefore we introduce a preliminary code simplification $\ulcorner\cdot\urcorner$ that takes care of ironing out the differences between the two languages. The translation $\ulcorner\cdot\urcorner$ acts on expressions, guards and codes, and it uses the companion $\lfloor\cdot\rfloor$ translation of environments into flat lists of variables. The simplifications $\ulcorner\cdot\urcorner$ and $\lfloor\cdot\rfloor$ are defined as follows[4]:

- the simplification of a formula φ, written $\ulcorner\varphi\urcorner$, is a homomorphic operator with respect to all arithmetic operators but subtraction and such that

$$\begin{array}{rcl}
\ulcorner e-e'\urcorner &=& \max(\ulcorner e\urcorner - \ulcorner e'\urcorner, 0)\\
\ulcorner x\urcorner &=& x \quad \text{if}\, x\notin Id\\
\ulcorner k\urcorner &=& k\\
\ulcorner m\in D\urcorner &=& \bigvee_{\chi\in fun(D)}(\ulcorner D.m\urcorner = \ulcorner D.\chi\urcorner)\\
\ulcorner m.payable\in D\urcorner &=& \bigvee_{\chi\in fun(D)}(\ulcorner D.m.p\urcorner = \ulcorner D.\chi\urcorner)\\
\ulcorner m\notin D\urcorner &=& \bigwedge_{\chi\in fun(D)}(\ulcorner D.m\urcorner \neq \ulcorner D.\chi\urcorner)\\
\ulcorner m.payable\notin D\urcorner &=& \bigwedge_{\chi\in fun(D)}(\ulcorner D.m.p\urcorner \neq \ulcorner D.\chi\urcorner)\\
\ulcorner fallback\in D &=& \mathtt{true}\ \text{if}\ D\ \text{declares the fallback function}, \mathtt{false}\ \text{otherwise}
\end{array}$$

where $\chi \in fun(D)$ is true if, for some function name m, $\chi = m$ and m is a non-payable function declared in D or if $\chi = m.\mathrm{p}$ and m is a payable function declared in D. The simplification on $D, D.m, D.m.\mathrm{p}$ can be picked to be any injective function whose codomain are integer values.

Moreover, $\ulcorner\cdot\urcorner$ also puts formulae φ in disjunctive normal form plus the additional constraint that atomic formulae are inequalities. For example, $e \neq e'$ is normalized to $e < e' \vee e' < e$.

- the flattening operation on environments Γ, noted $\lfloor\Gamma\rfloor$, encodes Γ into a list of integer expressions:

$$\begin{array}{ll}\text{let} \quad \Gamma & = [\ C_1 \mapsto [\mathtt{f}_{1,1} \mapsto e_{1,1}, \cdots, \mathtt{f}_{1,n_1} \mapsto e_{1,n_1}, balance \mapsto e_{1,b}],\\ & \cdots, C_k \mapsto [\mathtt{f}_{k,1} \mapsto e_{k,1}, \cdots, \mathtt{f}_{k,n_k} \mapsto e_{k,n_k}, balance \mapsto e_{k,b}]\ \ ]\end{array}$$

according to total orders $C_i \le C_{i+1}$ and $\mathtt{f}_{i,j} \le \mathtt{f}_{i,j+1}$, then

$$\lfloor\Gamma\rfloor \overset{\text{def}}{=} (\ulcorner e_{1,1}\urcorner, \cdots, \ulcorner e_{1,n_1}\urcorner, \ulcorner e_{1,b}\urcorner, \cdots, \ulcorner e_{k,1}\urcorner, \cdots, \ulcorner e_{k,n_k}\urcorner, \ulcorner e_{k,b}\urcorner)$$

Note that the flattening of a pure environment is a list of disjoint variables that can be used as formal parameters of a function.

- the simplification $\ulcorner\cdot\urcorner$ is lifted to the intermediate code as follows:

$$\begin{array}{rcl}
\ulcorner\Gamma\urcorner &=& \lfloor\Gamma\rfloor\\
\ulcorner e.m(\Gamma_1, \Gamma_2, e, \overline{e'}, D)\urcorner &=& e.m(\lfloor\Gamma_1\rfloor, \lfloor\Gamma_2\rfloor, \ulcorner e\urcorner, \ulcorner\overline{e'}\urcorner, \ulcorner D\urcorner)\\
\ulcorner\sum_{i\in I}(\phi)\ \Theta_i\urcorner &=& \sum_{i\in I}(\ulcorner\phi\urcorner)\ \ulcorner\Theta_i\urcorner
\end{array}$$

- the simplification $\ulcorner\cdot\urcorner$ of a program, i.e. a list of function definitions, is obtained simplifying each function in the list as follows:

$$\ulcorner C.m(\Gamma_0, \Gamma_1, v, \overline{x}, H) = \sum_{D\in Id}(H = D)\Theta_D\urcorner \overset{\text{def}}{=} C.m(\lfloor\Gamma_0\rfloor, \lfloor\Gamma_1\rfloor, v, \overline{x}, H) = \ulcorner\sum_{D\in Id}(H = D)\Theta_D\urcorner$$

A simplified intermediate program $Ł$ is turned into the sets of cost equation $\langle Ł\rangle$ as follows: every simplified function declaration

$$C.m(\lfloor\Gamma_0\rfloor, \lfloor\Gamma_1\rfloor, v, \overline{x}, H) = \sum_{i\in 1..h}\left(\bigvee_{j\in 1..k_i}\varphi_i^j\right)\Theta_i$$

(where each φ_i^j is a disjuction of comparisons between Presburger expressions) is turned into the following cost equations:

$$\begin{array}{lcll}
C.m(\lfloor\Gamma_0\rfloor, \lfloor\Gamma_1\rfloor, v, \overline{x}, H) &=& cost(\lfloor\Gamma_0\rfloor, \Theta_1) & [\phi_1^1]\\
&\ldots&&\\
C.m(\lfloor\Gamma_0\rfloor, \lfloor\Gamma_1\rfloor, v, \overline{x}, H) &=& cost(\lfloor\Gamma_0\rfloor, \Theta_1) & [\phi_1^{k_1}]\\
&\ldots&&\\
C.m(\lfloor\Gamma_0\rfloor, \lfloor\Gamma_1\rfloor, v, \overline{x}, H) &=& cost(\lfloor\Gamma_0\rfloor, \Theta_h) & [\phi_h^{k_1}]\\
&\ldots&&\\
C.m(\lfloor\Gamma_0\rfloor, \lfloor\Gamma_1\rfloor, v, \overline{x}, H) &=& cost(\lfloor\Gamma_0\rfloor, \Theta_h) & [\phi_h^{k_h}]
\end{array}$$

where

- *cost* is either $cost_{gain}^{C'}$ — to obtain the set of equations to compute the upper bound for the gain of C' — or $cost_{loss}^{C'}$ — to obtain the set of equations to compute the lower bound;
- $cost_{gain}^{C'}(\lfloor\Gamma\rfloor, \lfloor\Gamma'\rfloor) = \mathtt{max}(0, \Gamma'(C'.balance) - \Gamma(C'.balance))$ and

  $cost_{loss}^{C'}(\lfloor\Gamma\rfloor, \lfloor\Gamma'\rfloor) = \mathtt{max}(0, \Gamma(C'.balance) - \Gamma'(C'.balance))$;
- $cost(\lfloor\Gamma\rfloor, e.m(\lfloor\Gamma\rfloor, \lfloor\Gamma'\rfloor, \overline{e'})) = e.m(\lfloor\Gamma\rfloor, \lfloor\Gamma'\rfloor, \overline{e'})$ in both cases

Finally, if we are interested in the analysis of an invocation of the function $C.m$, we add a first equation

$$main(\lfloor\Gamma\rfloor, \overline{y}) = C.m(\lfloor\Gamma\rfloor, \lfloor\Gamma\rfloor, \overline{y}) \qquad [b_1 \ge 0 \wedge \ldots \wedge b_n \ge 0] \tag{1}$$

where $b_1, \ldots, b_n$ are the variables in $\lfloor\Gamma\rfloor, \overline{y}$ of type `uint` . We assume that these variables are non-negative (this is required because variables in `CoFloCo` are signed).

To conclude, if $\mathcal{I}$ is a program in the intermediate language and $C.m(\Gamma, \Gamma, z, \overline{x}, H)$ its initial state, then the Eq. 1 plus $\langle\ulcorner\mathcal{I}\urcorner\rangle$ gives the bunch of `CoFloCo` cost equations.

Example 11. To illustrate the output of our technique we compute the cost equations of the functions `Bank.pay` and `Thief.ack` in Example 3, according to the cost model that computes the loss of the `Bank`. We shorten `Bank` and `Thief` into B and T, respectively; for readability sake, we always write predicates such as $fallback \in T$ and $ack \in T$, even if the translator omits them because they evaluate to true (the functions belong to T). Similarly for the other predicates of the same shape. Equations whose guards are always false (e.g. $pay \in T$) are not shown nor generated by our translator.

[4] In the rest of the section we use the green color both for cost equations and for simplified intermediate programs, whose syntax is looser since it does not require formulae to be only conjunctions.

$$main(x_{B,b},x_{T,b},v,n,H)=B.pay(x_{B,b},x_{T,b},x_{B,b},x_{T,b},v,n,H)$$
$$[x_{B,b}\geq 0\wedge x_{T,b}\geq 0\wedge v\geq 0\wedge n\geq 0]$$
$$B.pay\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,n,H\big)=T.ack\big(x_{B,b},x_{T,b},y_{B,b}-n,y_{T,b}+n,0,B\big)$$
$$\big[H=T\wedge v\geq 1\wedge y_{B,b}>n\wedge n<5\wedge$$
$$fallback\in T\wedge ack\in T\big]$$
$$B.pay\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,n,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)\quad[H=T\wedge v<1]$$
$$B.pay\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,n,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)\quad\big[H=T\wedge y_{B,b}\leq n\big]$$
$$B.pay\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,n,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)\quad[H=T\wedge n\geq 5]$$
$$B.pay\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,n,H\big)=\mathtt{max}\big(0,x_{B,b}-\big(y_{B,b}-n+n\big)\big)$$
$$\big[H=B\wedge v\geq 1\wedge y_{B,b}>n\wedge n<5\wedge$$
$$fallback\in B\wedge ack\notin B\wedge ack.payable\notin B\big]$$
$$B.pay\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,n,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)\quad[H=B\wedge v<1]$$

$$B.pay\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,n,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)\qquad\big[H=B\wedge y_{B,b}\leq n\big]$$
$$B.pay\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,n,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)\qquad[H=B\wedge n\geq 5]$$
$$T.ack\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,H\big)=B.pay\big(x_{B,b},x_{T,b},y_{B,b}+1,y_{T,b}-1,1,2,T\big)$$
$$\big[H=B\wedge pay.payable\in B\wedge y_{T,b}\geq 1\big]$$
$$T.ack\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)$$
$$\big[H=B\wedge pay.payable\in B\wedge y_{T,b}<1\big]$$
$$T.ack\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)$$
$$\big[H=T\wedge pay\notin T\wedge pay.payable\notin T\wedge fallback\in T\wedge y_{T,b}\geq 1\big]$$
$$T.ack\big(x_{B,b},x_{T,b},y_{B,b},y_{T,b},v,H\big)=\mathtt{max}\big(0,x_{B,b}-y_{B,b}\big)$$
$$\big[H=T\wedge pay\notin T\wedge pay.payable\notin T\wedge fallback\in T\wedge y_{T,b}<1\big]$$

In Section 6 we analyze `CoFloCo` [9] outputs when these equations are fed to the tool.

The next theorem grants the correctness of our encoding according to the operational semantics for the (subset of) the syntax of `CoFloCo` we are considering. The proof is reported in the Appendix.

Theorem 12. (***Correctness of cost equation generation***) *Let $\mathcal{P}$ be an* `mSCL` *program such that $\vdash\mathcal{P}:\mathcal{I}$ and let $\mathcal{S}$ be an initial state and $\mathcal{S}\vdash C.m(\Gamma,\Gamma,v',\overline{v},H)$ and $C.m(\Gamma,\Gamma,v',\overline{v},H)$ $[xrArr]^{*}_{\mathcal{I}}\Gamma'$. Let us extend $\langle\ulcorner\mathcal{I}\urcorner\rangle$ (where we use either $cost^{C'}_{gain}$ or $cost^{C'}_{loss}$ during the translation) with a main function that calls C.m. Then*

1. *Determinism: $main(\lfloor\Gamma\rfloor,v',\overline{v},H)$ has a unique $\rightarrow_{\mathtt{CoFloCo}}$-normal-form*
2. *Correctness:*
   - *$main(\lfloor\Gamma\rfloor,v',\overline{v},H)\rightarrow_{\mathtt{CoFloCo}}{}^{*}\mathrm{GAIN}^{C'}_{\mathcal{I},C,m}(\Gamma,v',\overline{v},H)$ if we selected $cost^{C'}_{gain}$ during the translation,*
   - *$main(\lfloor\Gamma\rfloor,v',\overline{v},H)\rightarrow_{\mathtt{CoFloCo}}{}^{*}\mathrm{LOSS}^{C'}_{\mathcal{I},C,m}(\Gamma,v',\overline{v},H)$ if we selected $cost^{C'}_{loss}$ during the translation.*

The overall correctness of our technique is stated in Theorem 13. A preliminary statement about the correctness of our off-the-shelf tool `CoFloCo` is required. We have not found any such statement in the literature, therefore we conjecture it [9].

Conjecture 1. **(Correctness of `CoFloCo`)** Given a set of guarded cost equations whose first equation is

$$main(\overline{x})=m(\overline{e})\qquad[\varphi]$$

if `CoFloCo` claims that f is an upper bound to main on the domain where φ is true, then for every $\overline{x}$ belonging to such domain, $main(\overline{x})\leq f(\overline{x})$. When f is claimed to be an asymptotic bound, then $main\in\mathcal{O}(f)$ (on every $\overline{x}$ such that φ is true).

Theorem 13. (***Final theorem***) *Let $\mathcal{S}$ be an initial state of an* `mSCL` *program $\mathcal{P}$, $\vdash\mathcal{P}:\mathcal{I}$ and $\mathcal{S}\vdash C.m(\Gamma,\Gamma,v',\overline{v},H)$. If* `CoFloCo` *claims f to be an upper bound/an asymptotic upper bound to main of the equations obtained by $\langle\ulcorner\mathcal{I}\urcorner\rangle$ with a main function that calls C.m, then f is an upper bound/an asymptotic upper bound to $\mathtt{m}\mathrm{GAIN}_C(\mathcal{S})$, if $cost^{C}_{gain}$ was selected during the translation, or to $\mathtt{m}\mathrm{LOSS}_C(\mathcal{S})$ if $cost^{C}_{loss}$ was.*

Proof. Either $\mathcal{S}$ converges or it diverges. If it diverges, then both $\mathtt{m}\mathrm{gain}_C(\mathcal{S})$ and $\mathtt{m}\mathrm{LOSS}_C(\mathcal{S})$ are defined to be 0, and the statement trivially holds because all bounds computed by `CoFloCo` for our cost equations are non-negative.

If $\mathcal{S}$ converges then, by Theorem 2, the last reduction step leads to a final state $\mathcal{S}'$. Thus, by Definition 4 and Theorem 5, $C.m(\Gamma,\Gamma,v',\overline{v},H)$ $\Rightarrow^{*}_{\mathcal{I}}\Gamma'$ where $envs(\mathcal{S}')=\Gamma',\Gamma'$.

The thesis follows trivially from Corollary 1, Theorem 12 and Conjecture 1.

Note that `CoFloCo` may also compute a finite upper bound for diverging `mSCL` programs. This may sound strange because, usually, the cost equations fed to `CoFloCo` represent the computational cost in time of executing a program. Therefore, if the time is bounded, the program can not diverge. However, in our case, the cost equations compute the transfer of cryptocurrency. Hence, it is plausible to have a program that first transfers some asset and then enters into an infinite loop that does not change any balance. In this context, `CoFloCo` may compute a finite bound even if the program diverges. We recall that diverging computations of smart contracts are always aborted due to gas shortage and thus any non-negative bound is correct.

6. Assessments

We prototyped the cost analyzer of `mSCL` in about 2,500 lines of OCaml code. The code is then compiled to JavaScript to be run in the browser and can be found at the address: https://sacerdot.github.io/SmartAnalysis/behavioral_types. Our tool takes in input a list of smart contract declarations, produces a list of cost equations, and computes the cost equations of the first function of the first contract, say *C*. The user can choose between two cost models: the *gain* of *C*'s balance and the *loss* of *C*'s cryptocurrency. The cost equations can then be manually fed to `CoFloCo` to obtain an upper bound both in asymptotic form and in explicit form. Remarkably, the analyzer computes the worst scenario with respect to gaining and loosing because the computed cost depends on functions' input parameters, the initial value of all contracts' fields, including balances, and every possible caller.

The number of cost equations returned by our prototype is bi-linear in the number of functions and the number of contracts when the only variable of type `address` is `msg.sender`. (It is worth noticing that, in the following examples, we have used the extension of the prototype that also deals with `address` data types, see Section 7, which makes the number of cost equations exponential with respect to `address` variables, where the base is the number of contracts, see Section 9).

To test the tool and gain preliminary experience, we have analyzed several smart contracts from `etherscan.io`. This required little programming overhead for most of the contracts in order to rewrite Solidity code in `mSCL`. In Table 1, we report our analysis of four archetypal contracts we identified among the other ones. In particular, for every program, we

Table 1
Statistics on a few archetypal examples.

	♯ LOC	♯ LMC	♯ Equations	CoFloCo's Time for gain + loss
Bank-Thief code	20	20	20	734 ms + 240 ms
English Auction Scheme	32	32	38	509 ms + 468 ms
Handover Ponzi Scheme	42	50	336	6,964 ms + 4,784 ms
Chain-shaped Ponzi scheme	45	63	1030	27,978 ms + 27,962 ms

give the lines of original code (♯ LOC), those produced by our translation (♯ LMC), the number of equations produced (♯ Equations) and the sum of CoFloCo times for computing the upper bound to the gain and to the loss.

Few remarks about the output of the analyzer are in order. In the Bank-Thief code of Fig. 1, the costs are a function of the initial values of Bank's balance (Bank_ _balance_), Thie's balance (Thief_ _balance_), the invoker of the analyzed function (_msg_sender_), the amount of coins passed to the function (_msg_value_) and the function parameter (N). CoFloCo's output is:

```
MAXIMUM GAIN:
Maximum cost of main__(Bank__balance_,Thief__balance_,
     _msg_sender_,_msg_value_,N):
    nat(-Bank__balance_+2)
Asymptotic class: n

MAXIMUM LOSS:
Maximum cost of main__(Bank__balance_,Thief__balance_,
     _msg_sender_,_msg_value_,N):
    nat(Bank__balance_-2)
Asymptotic class: n
```

where nat(x) returns the maximum between x and 0. The output shows that the attack can be successful: the bank can lose all of its balance, but for 2 coins. It can also happen that the bank earns money instead, but only up to 2. This happens when the initial bank account has fewer than two coins. A careful analysis by the hand of the code tells us that the upper bound to the loss computed by CoFloCo is tight, while the one for the gain is not: the bank can actually only win one coin.

In the English Auction Scheme, the smart contract Auctioneer records the address of the bidder that is currently winning the auction, together with his bid. When a new bid arrives, if it is greater than the current winning one, the previous winner is refunded. Otherwise, the bid is refunded to the sender. The result of the analysis is interesting:

```
MAXIMUM GAIN:
Maximum cost of main__(Bidder1__balance_,Auctioneer__balance_,
     Auctioneer_max,Bid1, ...): 0
Asymptotic class: constant

MAXIMUM LOSS:
Maximum cost of main__(Bidder1__balance_,Auctioneer__balance_,
     Auctioneer_max,Bid1, ...): max([nat(Bid1),
     nat(-Auctioneer_max+Bid1)])
Asymptotic class: n
```

The bidder cannot gain any money by bidding: either he can lose all its bid nat(Bid1) (because he is winning the auction) or he can lose the lesser amount nat(-Auctioneer_max+Bid1) because he was already winning and decided to lift his offer (the previous offer is returned back).

In the “Handover Ponzi scheme” [11], every user invests more money than the current price and he receives back more money than the amount invested when the next user joins the scheme. The current price is augmented (by 50%) every time a new user joins in order to provide an income to all users. The 10% of the money invested by every user is reclaimed by the owner of the contract and thus only 90% is used to pay the previous user.

We analyze two scenarios. The first scenario is when Player joins the scheme, followed by Player2. The analysis yields:

```
MAXIMUM GAIN:
Maximum cost of main__(Player__balance_,Player_amount,
     Player2__balance_,N, ...): nat(7/20*N)
Asymptotic class: n

MAXIMUM LOSS:
Maximum cost of main__(Player__balance_,Player_amount,
     Player2__balance_,N, ...): 0
Asymptotic class: constant
```

In this scenario Player does not lose money and it can gain $\frac{7}{20}$ of the invested money. An analysis by hand shows that the bound is tight[5] and it is both an upper and a lower bound. Note that it is not trivial to figure out the fraction $\frac{7}{20}$ just looking at the code where the only constants that occur are $\frac{9}{10}$ and $\frac{3}{2}$.

In the second scenario, Player is the unique player. The analysis yields:

```
Maximum cost of main__(Player__balance_,Player_amount,
     N,...): 0
Asymptotic class: constant

MAXIMUM LOSS:
Maximum cost of main__(Player__balance_,Player_amount,
     N,...): nat(N)
Asymptotic class: n
```

In this scenario Player loses all the money he invested.

The remarks about the outputs of the Chain-shaped Ponzi scheme are omitted because similar to the Handover Ponzi scheme.

7. Extensions of the analysis

Three features, which are relevant for the expressivity of mSCL, have not yet been discussed: (i) address and bool data types, (ii) functions invocations with explicit continuations, and (iii) dynamic creation of smart contracts and their deployment. The first two have already been integrated into our analyzer. We discuss these extensions in this section.

7.1. Addresses

The extension of the encoding in Section 5 to cope with addresses is not difficult. Indeed, it is sufficient to follow the same scheme we used to deal with msg.sender that was the only parameter of type address . In particular, the translation rule for functions whose formal parameters are also addressed becomes

[5] The first user pays x; the second one must pay $\frac{3}{2}x$, and 90% of it, i.e. $\frac{3}{2}\frac{9}{10}x$ goes back to the first user whose final gain is $\frac{3}{2}\frac{9}{10}x - x = \frac{27}{20}x - x = \frac{7}{20}x$.

[FUNCTION-ADDR]
$$\dfrac{\begin{array}{c}\left(\begin{array}{l}\Gamma_0(D') = [\mathtt{f}_1 \mapsto x_{D',1}, \cdots, \mathtt{f}_n \mapsto x_{D',n}, \mathit{balance} \mapsto x_{D',b}] \\ \Gamma_1(D') = [\mathtt{f}_1 \mapsto y_{D',1}, \cdots, \mathtt{f}_n \mapsto y_{D',n}, \mathit{balance} \mapsto y_{D',b}]\end{array}\right)^{\{\mathtt{f}_1,\cdots,\mathtt{f}_n,\mathit{balance}\}=\mathit{fields}(D'),\, D'\in Id} \\ \mathtt{function\ m}(\overline{\mathsf{T}\ x}, \overline{\mathtt{address}\ \ z})[\mathtt{payable}]\{\overline{\mathsf{T}\ y;}\ \mathsf{S}\} \in C \\ \left(\Gamma_0, \Gamma_1[\overline{x} \mapsto \overline{u}, \overline{z} \mapsto \overline{D}, \overline{y} \mapsto \overline{0}] \vdash^{v}_{C',C} \mathsf{S} : \Theta_{C',\overline{D}}\right)^{C',\overline{D}\in Id}\end{array}}{\Gamma_0, \Gamma_1 \vdash C.m(\Gamma_0,\Gamma_1,v,\overline{u},\overline{D'},H) = \sum_{C'\in Id}(H{=}C')\sum_{\overline{D'}\in Id}(\overline{D'}{=}\overline{D})\ \Theta_{C',\overline{D}}}$$

(for readability sake we have separated addresses from other types). That is, `address` variables add (finite) alternatives in the body of functions in order to cope with every possible instance of the variable.

Once we make the address type a first class citizen, we also have to deal with local variables and field names that store addresses. The solution remains the same: the translation of each function body must start with a nested sum for each field, parameter, and local variable of type address, where each summand differs from the previous one by the value taken by the variable in the finite set of known contract addresses.

7.2. Booleans

Booleans are encoded in the intermediate language using 0 (for false) and 1 (for true). The occurrence of a boolean variable/parameter/field b in an expression is encoded as $b = 1$.

Assignment to boolean variables and invocation of a function that takes a boolean argument is slightly annoying because a boolean expression (such as b_1 *&&* b_2) can not be directly encoded as an arithmetic expression in the usual way ($b_1 * b_2$) because of the restrictions due to Presburger arithmetics. This issue is solved by introducing a conditional statement for every assignment/actual parameter. E.g. $x = b_1$ *&&* b_2; is equivalent to `if` (b_1 *&&* b_2) { $x = 1$; } `else` { $x = 0$; }. Therefore the intermediate code will have one additional binary sum for each assignment to a boolean value and for each boolean expression in a function call.

7.3. Continuations

Dealing with explicit continuations of function invocations is not straightforward because our intermediate language in Section 4 only admits tail-recursive (or tail mutual recursive) invocations. Nevertheless, the extension of the analyzer with explicit continuations is significant because it allows one to verify fallback functions with non-empty bodies.

To illustrate our solution, consider the following extension of the `mSCL` syntax:

$$\begin{array}{ll}
\mathsf{C} ::= & \mathtt{contract}\ C\ \{\ \ \overline{\mathsf{T}\ \mathtt{f};}\ \ \mathsf{F}\ \ [\mathtt{fallback(\,)}\ \mathtt{payable}\ \{\overline{\mathsf{T}\ x;}\ \mathsf{S}\}]\ \} \\
\\
\mathsf{F} ::= & \cdots\ \mid\ \mathtt{function}\ \mathtt{m}(\overline{\mathsf{T}x})[\mathtt{payable}]\ \mathtt{returns}\ (\mathsf{T})\ \{\overline{\mathsf{T}\ y;}\ \mathsf{S}\}\ \ \ \mathsf{F} \\
\\
\mathsf{S} ::= & \cdots\ \mid\ \mathtt{if}\ (\mathsf{E})\ \{\ \mathsf{S}\ \}\ \mathtt{else}\ \{\ \mathsf{S}\ \};\ \mathsf{S}\ \mid\ \mathtt{return}(\mathsf{E}) \\
& \mid \mathtt{return}(\mathsf{E}.\mathtt{m}[.\mathtt{value}(\mathsf{E})](\overline{\mathsf{E}})) \\
& \mid\ \ \mathsf{E}.\mathtt{m}[.\mathtt{value}(\mathsf{E})](\overline{\mathsf{E}});\ \mathsf{S}\ \mid\ x = \mathsf{E}.\mathtt{m}[.\mathtt{value}(\mathsf{E})](\overline{\mathsf{E}});\ \mathsf{S}
\end{array}$$

In this extended syntax fallback functions may now have non-empty bodies; function bodies may also return values; function invocations, as well as conditionals, may have continuations.

The translation of the above language expands the one in Section 4 by using the standard CPS translation for removing continuations. Actually, there is one difference: instead of using a higher order language (which is required by CPS), we keep the same intermediate language by extending functions' arguments with another one representing the stack of activation records for continuations. However, since the arity of `CoFloCo` functions is fixed, we won't be able to manage stacks of arbitrary size. We solve the issue by parameterizing our translation with a constant κ that limits the length of the stack. The value κ can be chosen as follow:

1. compute the graph of invocations where nodes correspond to function definitions and arcs to function calls;
2. assign weight 0 to the arcs that correspond to tail invocations and 1 to the other arcs; (Tail invocations have weight 0 because, to preserve expressivity, our translation implements the tail (mutual) recursion optimization, so that tail calls do not require more stack space. In particular, programs whose functions are all tail-recursive require only one frame for the initial call.)
3. (a) if the graph of invocation contains no cycles of unbound weight, choose κ as the maximal weight of a path;
   (b) otherwise the value for κ is requested to the user by the analyzer. In this case, we are technically verifying the κ-th approximant of the program, which reverts if it tries to nest more than κ non-tail-recursive calls, exhausting the stack space.

Let ι be the maximum number of parameters and local variables of functions. The maximal size of the stack is bounded by $(\iota + 4) \times (\kappa + 1)$ where $\iota + 4$ is the maximal size of a frame (the 4 is due to the extra slots in the stack frame to record the function caller, the callee, the function identifier and the amount of cryptocurrency transferred). The extra unit added to κ is required for the additional stack frame used for the initial call to the program.

More formally, let σ be a sequence of frames of the form

$$\underbrace{\alpha_1 \cdots \alpha_h \perp\!\!\!\perp \cdots \perp\!\!\!\perp}_{\kappa+1\ \text{frames}}$$

where

- the initial frames α_i are equal to $\langle C, \mathtt{m}, \overline{e}, D, e' \rangle$, where C is the callee, $\mathtt{m}$ is callee's function to be executed with arguments $\overline{e}$, assuming it has been called by D that has transferred e' cryptocurrencies;
- the frames $\perp\!\!\!\perp$, called *empty frames*, are equal to $\langle \perp, \mathtt{m}_\perp, \overline{\perp}, \perp, 0 \rangle$, where $\perp$, $\mathtt{m}_\perp$ are special names and we assume $\perp$ to be a new valid expressions;
- there is no empty frame to the left of a non-empty frame, and the last (i.e. rightmost) frame in σ is always empty;
- a frame $\langle C, \mathtt{m}, \overline{e}, D, e' \rangle$ is equal to $\perp\!\!\!\perp$ if and only if $C = \perp$.

A sequence σ represents the stack of continuations: every non-empty frame stands for a continuation to be executed; the first empty frame denotes the end of the stack. We use the following operations on tuples of expressions and on sequences of frames:

- $|\overline{e}|$ returns the length of the tuple;
- $\overline{e}{\downarrow_{D.\mathtt{m}}}$ returns the prefix of $\overline{e}$ whose length is equal to the number of arguments of $D.\mathtt{m}$;
- $\sigma^{\Rsh}$ drops the last frame in σ, *i.e.* $(\sigma' \alpha)^{\Rsh} = \sigma'$;
- $[\sigma]^{\looparrowright}$ returns the first element of the last-but-one frame in σ, if it exists, or any value different from $\perp$, if $k = 0$. That is $[\sigma' \langle C, \mathtt{m}, \overline{e}, D, e' \rangle \perp\!\!\!\perp]^{\looparrowright} = C$. The key property is that if $[\sigma]^{\looparrowright} = \perp$ then the last-but-one frame is

[ASGN-CONT-INVK]

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in \mathit{Id} \quad [\Gamma_1 \vdash^{e}_{C,D} \mathsf{E}' : e'] \quad \Gamma_1 \vdash^{e}_{C,D} \overline{\mathsf{E}''} : \overline{e''} \\ \mathit{dom}(\Gamma_1|_{\mathit{Var}}) = \overline{w} \quad \texttt{function}\ \mathtt{m}_{\mathsf{S}}(\mathsf{T}'\ x', \overline{\mathsf{T}_w\ w})\ \texttt{payable}\ \{x{=}x';\mathsf{S}\} \in D \\ \Gamma_1(\overline{w}) = \overline{e_w} \qquad |\overline{\bot}| = \iota - |\overline{e_w}| \qquad \sigma' = \langle D,\mathtt{m}_{\mathsf{S}},\overline{e_w}\overline{\bot},C,e\rangle\ \sigma^{\Rsh} \\ \Gamma_0, \Gamma_1 \vdash^{e,\sigma'}_{C,D} e_0.\mathtt{m}[.\mathtt{value}(e')](\overline{e''}) : \Theta\end{array}}{\Gamma_0, \Gamma_1 \vdash^{e,\sigma}_{C,D} x{=}\mathsf{E}.\mathtt{m}[.\mathtt{value}(\mathsf{E}')](\overline{\mathsf{E}''});\mathsf{S} : ([\sigma]^{\looparrowright}{=}\bot)\ \Theta + ([\sigma]^{\looparrowright}{\neq}\bot)\ \Gamma_1}$$

[INVK-CONT]

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in \mathit{Id} \quad [\Gamma_1 \vdash^{e}_{C,D} \mathsf{E}' : e'] \quad \Gamma_1 \vdash^{e}_{C,D} \overline{\mathsf{E}''} : \overline{e''} \\ \mathit{dom}(\Gamma_1|_{\mathit{Var}}) = \overline{w} \quad \texttt{function}\ \mathtt{m}_{\mathsf{S}}(\overline{\mathsf{T}_w\ w})\ \texttt{payable}\ \{\mathsf{S}\} \in D \\ \Gamma_1(\overline{w}) = \overline{e_w} \qquad |\overline{\bot}| = \iota - |\overline{e_w}| \qquad \sigma' = \langle D,\mathtt{m}_{\mathsf{S}},\overline{e_w}\overline{\bot},C,e\rangle\ \sigma^{\Rsh} \\ \Gamma_0, \Gamma_1 \vdash^{e,\sigma'}_{C,D} e_0.\mathtt{m}[.\mathtt{value}(e')](\overline{e''}) : \Theta\end{array}}{\Gamma_0, \Gamma_1 \vdash^{e,\sigma}_{C,D} \mathsf{E}.\mathtt{m}[.\mathtt{value}(\mathsf{E}')](\overline{\mathsf{E}''});\mathsf{S} : ([\sigma]^{\looparrowright}{=}\bot)\ \Theta + ([\sigma]^{\looparrowright}{\neq}\bot)\ \Gamma_1}$$

[INVK-TAIL-NV]

$$\frac{\Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in \mathit{Id} \quad \Gamma_1 \vdash^{e}_{C,D} \overline{\mathsf{E}} : \overline{e'}}{\Gamma_0, \Gamma_1 \vdash^{e,\sigma}_{C,D} \mathsf{E}.\mathtt{m}(\overline{\mathsf{E}}) : \begin{array}{ll} & (m{\in}e_0)\ e_0.m(\Gamma_0,\Gamma_1,0,\overline{e'},D)\sigma \\ + & (m.\mathit{payable}{\in}e_0)\ e_0.m(\Gamma_0,\Gamma_1,0,\overline{e'},D,\sigma) \\ + & (m{\notin}e_0 \wedge m.\mathit{payable}{\notin}e_0 \wedge \mathit{fallback}{\in}e_0)\ e_0.\mathit{fallback}(\Gamma_0,\Gamma_1,0,D,\sigma) \\ + & (m{\notin}e_0 \wedge m.\mathit{payable}{\notin}e_0 \wedge \mathit{fallback}{\notin}e_0)\ \Gamma \end{array}}$$

[INVK-TAIL]

$$\frac{\begin{array}{c}\Gamma_1 \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in \mathit{Id} \quad \Gamma_1 \vdash^{e}_{C,D} \overline{\mathsf{E}} : \overline{e'} \quad \Gamma_1 \vdash^{e}_{C,D} \mathsf{E}' : e'' \\ \Gamma_1' = \Gamma_1[e_0.\mathit{balance} \mapsto^{+} e''][D.\mathit{balance} \mapsto^{-} e'']\end{array}}{\begin{array}{l}\Gamma_0, \Gamma_1 \vdash^{e,\sigma}_{C,D} \mathsf{E}.\mathtt{m}.\mathtt{value}(\mathsf{E}')(\overline{\mathsf{E}}) : \\ \qquad \begin{array}{ll} & (m{\in}e_0)\ \Gamma \\ + & (m.\mathit{payable}{\in}e_0 \wedge \Gamma_1(D.\mathit{balance}){\geqslant}e'')\ e_0.m(\Gamma_0,\Gamma_1',e'',\overline{e'},D,\sigma) \\ + & (m.\mathit{payable}{\in}e_0 \wedge \Gamma_1(D.\mathit{balance}){<}e'')\ \Gamma \\ + & (m{\notin}e_0 \wedge m.\mathit{payable}{\notin}e_0 \wedge \mathit{fallback}{\in}e_0 \wedge \Gamma_1(D.\mathit{balance}){\geqslant}e'') \\ & \hfill e_0.\mathit{fallback}(\Gamma_0,\Gamma_1',e'',D,\sigma) \\ + & (m{\notin}e_0 \wedge m.\mathit{payable}{\notin}e_0 \wedge \mathit{fallback}{\in}e_0 \wedge \Gamma_1(D.\mathit{balance}){<}e'')\ \Gamma_0 \\ + & (m{\notin}e_0 \wedge m.\mathit{payable}{\notin}e_0 \wedge \mathit{fallback}{\notin}e_0)\ \Gamma \end{array}\end{array}}$$

[FUNCTION-CONT]

$$\frac{\begin{array}{c}\left(\begin{array}{c}\Gamma_0(D) = [\mathtt{f}_1 \mapsto x_{D,1}, \cdots, \mathtt{f}_n \mapsto x_{D,n}, \mathit{balance} \mapsto x_{D,b}] \\ \Gamma_1(D) = [\mathtt{f}_1 \mapsto y_{D,1}, \cdots, \mathtt{f}_n \mapsto y_{D,n}, \mathit{balance} \mapsto y_{D,b}]\end{array}\right)^{\{\mathtt{f}_1,\cdots,\mathtt{f}_n,\mathit{balance}\}=\mathit{fields}(D), D\in \mathit{Id}} \\ \texttt{function}\ m(\overline{\mathsf{T}_x\ x})[\texttt{payable}]\{\overline{\mathsf{T}_y\ y};\mathsf{S}\} \in D \\ \sigma = \langle z_{1,1}, \cdots, z_{1,\iota+4}\rangle \cdots \langle z_{\kappa,1}, \cdots, z_{\kappa,\iota+4}\rangle \perp\!\!\!\perp \quad \left(\Gamma_0, \Gamma_1[\overline{x} \mapsto \overline{x_0}, \overline{y} \mapsto \overline{\bot}] \vdash^{v,\sigma}_{C,D} \mathsf{S} : \Theta_C\right)^{C\in \mathit{Id}}\end{array}}{\Gamma_0, \Gamma_1 \vdash D.m(\Gamma_0,\Gamma_1,v,\overline{x_0},H,\sigma) {=} \sum_{C\in \mathit{Id}}(H{=}C)\ \Theta_C}$$

Fig. 7. Translation of mSCL statements with continuations, Part II.

unused (it is $\perp\!\!\!\perp$), therefore the sequence σ is not full and it can hold one more element. It is sufficient to throw away the last empty frame (using $\cdot^{\Rsh}$) and push the new element at the beginning, effectively shifting to the right all the already present frames.

Finally, we write $\mathtt{m} \in \mathit{fun}(C)$ to mean that $C \in \mathit{cnames}(\mathcal{P})$ and the smart contract named C has a function m. The notation keeps $\mathcal{P}$ implicit.

Figs. 6 and 7 report the rules for translating mSCL programs in intermediate codes. Rule [EMPTY-CONT] deals with empty statements. It has two subcases, according to the sequence σ is of the form $\perp\!\!\!\perp \cdots \perp\!\!\!\perp$ – the stack is empty or not. According to our modeling, the former case is when $H = \bot$, where H is the first element of the initial frame. In this case, we return to the current environment. In the second case, we evaluate the continuation, say D.m (in the judgment in the premise we drop out useless expressions in the initial frame). That is, in our translation, continuations of invocations and conditionals are managed by ad-hoc functions that extend those of the mSCL program. Each one of these new functions implements a continuation. Since the continuation can access `msg.sender` and `msg.value`, it is important that when the continuation is called the right values are restored: this is achieved by making these functions `payable` and by translating in the premise D'.m.value(e').(…) that sets `msg.sender` to D' and `msg.value` to e'. However, this also transfers again e' units of cryptocurrency from `msg.sender` to the receiver (see rule [INVK-TAIL]), which would not be correct. To contrast this, we perform the translation in Γ_1' where we first transfer back e' units from the receiver to `msg.sender`.

We notice that in the premise of [EMPTY-CONT] we pass two expressions of the intermediate language (e' and e'') where expressions of mSCL are expected. This is correct since, according to the rules in Fig. 3, e' and e'' are expressions in the source language mSCL as well.

Rule [RETURN-CONT] defines the code for return statements; it has similar premises to [EMPTY-CONT], except for the return value. We notice that, in this case, the continuation stored in the sequence of frames lacks the first argument that is provided by the return and is therefore taken by $\mathtt{m}_S$. Rules [ASGN-CONT-INVK] and [INVK-CONT] define invocations with continuations when invocations return a value or when the function is void. In both cases, we assume the functions of the corresponding smart contract are extended with new ad-hoc functions managing the continuation. The formal parameters of these ad-hoc functions are the variables in the current environments (e.g. $dom(\Gamma_1|_{Var})$, see also rule [FUNCTION-CONT]) plus an additional variable for functions returning a value. In this case, the sequence of frames stores the values of the variables in the environment, which will be restored when the continuation is triggered (see rules [EMPTY-CONT] and [RETURN-CONT]).

Rules [INVK-TAIL-NV] and [INVK-TAIL] extend rules [INVK-NV] and [INVK] of Fig. 4 by taking into account stacks. We notice that our translation implements the tail (mutual) recursion optimization, so that tail calls do not require more stack space.

Example 14. The extension of mSCL with continuations allows us to write the code of a DAO-like attack [4]:

```
contract Bank {
    uint to_pay=5 ;
    function pay(uint n) {
        if (this.balance>n && n≤to_pay) {
            msg.sender.transfer(n) ;
            to_pay=to_pay-n ;
        }
    }
}

contract Thief {
    fallback() payable{
        msg.sender.pay(1) ;
    }
}
```

The contract `Bank` admits withdraws of at most `to_pay` cryptocurrencies, provided that the account balance is large enough. However, the `Thief` client circumvents this constraint by exploiting a feature of mSCL *(*and of Solidity*)* according to which `Thief`*'s* fallback function is invoked when it is the recipient of a `transfer`. In fact, in the above code, `Thief`*'s fallback* contains an invocation to `Bank`*'s* `pay` function that is performed without having updated the `to_pay` field. It turns out that the overall effect of an invocation `Bank.pay(1)` by `Thief` is to drain the account.

Note that the graph of invocations is cyclic; therefore our technique analyzes the κ-th approximant and computes the corresponding maximal gains and losses. In these cases, we actually compute two consecutive approximants and deduce properties of the code according to the differences in the results. The following intermediate code defines the approximant 1 of the DAO-like attack.

Let Γ_0 and Γ_1 two pure environments (with disjoint codomains) defined as follows:

$$\begin{aligned}\Gamma_0 &= \big[Bank \mapsto \big[balance \mapsto x_{Bank,b}; to\_pay \mapsto x_{Bank,to\_pay}\big],\\ &\qquad Thief \mapsto \big[balance \mapsto x_{Thief,b}\big]\ \big]\\ \Gamma_1 &= \big[Bank \mapsto \big[balance \mapsto y_{Bank,b}; to\_pay \mapsto y_{Bank,to\_pay}\big],\\ &\qquad Thief \mapsto \big[balance \mapsto y_{Thief,b}\big]\ \big]\end{aligned}$$

To improve readability, we use the following abbreviations and conventions:

- $\sigma = s_1, \ldots, s_{12}$,
- we hide dead code, i.e. code that will never be executed. The detection of dead code has been performed by hand, but this can be automatized using standard techniques (e.g. abstract interpretation);
- H is always the value of `msg.sender` and v the value of `msg.value`*;*
- in *Bank.pay*:

$$\begin{aligned}\Gamma_1' &= \Gamma_1[Bank.balance \mapsto^{-} n, Thief.balance \mapsto^{+} n]\\ \sigma' &= pay\_cont, v, Bank, n, 0, s_1, s_2, s_3, s_4, s_5, s_6\end{aligned}$$

The intermediate code for *Bank.pay* is (comments are added for readability sake):

```
$Bank.pay(\Gamma_0,\Gamma_1,H,v,n,\sigma)=$
 $(H=Bank)\ldots$ --dead code: the Bank never calls Bank.pay
 $+(H=Thief)$
  $(y_{Bank,b}>n \wedge y_{Bank,to\_pay} \geq n)$ --if(this.balance>n && n <=to_pay)
    $(s1=\perp)$ --stack not full
     $(n\geq 0 \wedge y_{Bank,b} \geq n)$ --enough money to transfer
       --$\sigma'$=<C=Bank, m=pay_cont, e=msg.value, D=Bank, e'=n, 0>$\sigma$
     --where pay_cont implements continuation to_pay=to_pay−n;
     --msg.sender.transfer() calls Thief.fallback
       $Thief.fallback(\Gamma_0,\Gamma_1',Bank,n,Bank,\sigma')$
     $+(n<0 \wedge y_{Bank,b} \geq n)\Gamma_1$ --revert: not enough money to transfer
    $+(s_1 \neq \perp)\Gamma_1$ --revert: push on full stack
 $+(!(y_{Bank,b}>n \wedge y_{Bank,to\_pay} \geq n))$ --else case: pop and call next continuation
   $runtime\_dispatch(\Gamma_0,\Gamma_1,0,0,0,\sigma)$
```

The function pay_cont implementing the continuation `to_pay = to_pay - n`; is:

$$\begin{aligned}&pay\_cont(\Gamma_0, \Gamma_1, H, v, r, n, \sigma)\\ &\qquad = runtime\_dispatch(\Gamma_0, \Gamma_1[Bank.to\_pay \mapsto^{-} n], 0, 0, 0, \sigma)\end{aligned}$$

where runtime_dispatch defines the code that pops a continuation from the stack and calls it, or it commits the computation when the stack is empty. We let

$$\sigma'' = s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, \perp^6 .$$

```
$runtime\_dispatch(\Gamma_0,\Gamma_1,H,v,r,\sigma)=(s_1=Bank)$
   $s_2=pay\_cont)\ pay\_cont(\Gamma_0,\Gamma_1,s_4,s_3,r,s_5,\sigma'')$    $+\ (s_2=pay)\ldots$
      --Bank.pay is never used as a continuation
      $+(s_1=Thief)(s_2=fallback)\ldots$ --Thief.fallback is never
        used as a continuation
      $+(s_1=\perp)\,\Gamma_1$ --empty continuation stack: commit
```

Finally, the intermediate code for Thief.fallback is:

```
$Thief.fallback(\Gamma_0,\Gamma_1,H,v,\sigma)=$
  $(H=Bank)\ Bank.pay(\Gamma_0,\Gamma_1,Thief,0,1,\sigma)$ --tail call to Bank.pay
  $+(H=Thief)\ldots$ --the Thief never calls its fallback
```

The code for the second approximant is the same, with the following minor changes:

- $\sigma = s_1, \ldots, s_{18}$
- $\sigma' = pay\_cont$, v, *Bank*, n, 0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}
- $\sigma'' = s_7$, s_8, s_9, s_{10}, s_{11}, s_{12}, s_{13}, s_{14}, s_{15}, s_{16}, s_{17}, s_{18}, $\perp^6$.
- the stack not full check in the *Bank.pay* equation becomes $(s_7 = \perp)$

7.4. The analysis

We use the same arguments as in Section 5 to make the intermediate code of the above translation adequate to a cost analyzer. In particular, we conform to the same cost models and below we only detail the differences due to the need of passing around the encoding of the stack.

Let $(\mathsf{C}_1, \cdots, \mathsf{C}_n)$ be a program in the `mSCL` extended syntax, where κ is the maximal weight of a path in the graph of invocations and ι is the maximal number of arguments and local variables of a function. For every sequence of $\kappa + 1$ frames $\sigma = \alpha_1 \cdots \alpha_h \perp\!\!\perp \cdots \perp\!\!\perp$, let $\overparen{\sigma}$ be the tuple whose length is $(\iota + 4) \times (\kappa + 1)$ that is defined as follows:

$$\overparen{\alpha_1 \cdots \alpha_h \perp\!\!\perp \cdots \perp\!\!\perp} \;=\; \overparen{\alpha_1}, \cdots, \overparen{\alpha_h}, \overparen{\perp\!\!\perp}, \cdots, \overparen{\perp\!\!\perp}$$

$$\overparen{\langle C, m, \overline{e}, D, e' \rangle} \;=\; C, m, \overline{e}, \underbrace{\perp, \cdots, \perp}_{\iota - |\overline{e}| \text{ times}}, D, e'$$

$$\overparen{\perp\!\!\perp} \;=\; \underbrace{\perp, \cdots, \perp}_{\iota + 4 \text{ times}}$$

Sequences of $(\iota + 4) \times (\kappa + 1)$ elements will be ranged over by σ. Cost equations of an `mSCL` program are derived from the corresponding intermediate code as follows:

1. for every function $C.m(\Gamma_0, \Gamma_1, v, \overline{x}, H, \sigma) = \Theta_{C.m}$, let $\bigvee_{i \in 1..h} (\varphi_i)\ \Theta_i$ be the canonical form of $\ulcorner \Theta_{C.m} \urcorner$ (therefore every φ_i is a conjunction). Then we have the following cost equations:

$$\begin{array}{rcll} C.m(\lfloor \Gamma_0 \rfloor, \lfloor \Gamma_1 \rfloor, v, \overline{x}, H, \overparen{\sigma}) & = & \Theta_1 & [\varphi_1] \\ & \cdots & & \\ C.m(\lfloor \Gamma_0 \rfloor, \lfloor \Gamma_1 \rfloor, v, \overline{x}, H, \overparen{\sigma}) & = & \Theta_h & [\varphi_h] \end{array}$$

2. if we are interested in the analysis of an invocation of the function $C.m$, we add the next equation where we initialize the stack to an empty one:

$$\mathit{main}(\lfloor \Gamma \rfloor, \overline{y}) = C.m(\lfloor \Gamma \rfloor, \lfloor \Gamma \rfloor, \overline{y}, \overparen{\perp\!\!\perp \cdots \perp\!\!\perp}) \qquad [b_1 \geqslant 0 \wedge \ldots \wedge b_n \geqslant 0]$$

where $b_1, \ldots, b_n$ are the variables in $\lfloor \Gamma \rfloor, \overline{y}$ of type `uint` . We assume that these variables are non-negative (this is required because variables in `CoFloCo` are signed).

Example 15. The cost equations of the first approximant that are generated by our analyzer for the functions in Example 14 are the following ones. We use the abbreviations

$$\begin{array}{rcl} \perp^n & = & \underbrace{\perp, \cdots, \perp}_{n \text{ times}} \\ \sigma & = & s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12} \\ \sigma{\uparrow}^6 & = & s_7, s_8, s_9, s_{10}, s_{11}, s_{12}\ . \end{array}$$

$$\begin{gathered}
\mathit{main}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, H, v, n) \\
= \mathit{bank\_pay}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, H, v, n, \perp^{12}) \\
[x_{Bank,b} \geq 0 \wedge x_{Thief,b} \geq 0 \wedge v \geq 0 \wedge x_{Bank,to\_pay} = 5] \\
\mathit{bank\_pay\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, n, \sigma) \\
= \mathit{thief\_fallback}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b} - n, y_{Bank,to\_pay}, y_{Thief,b} + n, \\
\mathit{Bank}, n, \mathit{Bank}, \mathit{pay\_cont\_int\_int}, v, \mathit{Bank}, n, 0, s_1, s_2, s_3, s_4, s_5, s_6) \\
[H = \mathit{Thief} \wedge y_{Bank,b} > n \wedge y_{Bank,to\_pay} \geq n \wedge s_1 = \perp \wedge n \geq 0 \wedge y_{Bank,b} \geq n] \\
\mathit{bank\_pay\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, n, \sigma) = 0 \\
[H = \mathit{Thief} \wedge y_{Bank,b} > n \wedge y_{Bank,to\_pay} \geq n \wedge s_1 = \perp \wedge n < 0] \\
\mathit{bank\_pay\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, n, \sigma) = 0 \\
[H = \mathit{Thief} \wedge y_{Bank,b} > n \wedge y_{Bank,to\_pay} \geq n \wedge s_1 = \perp \wedge y_{Bank,b} < n] \\
\mathit{bank\_pay\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, n, \sigma) = 0 \\
[H = \mathit{Thief} \wedge y_{Bank,b} > n \wedge y_{Bank,to\_pay} \geq n \wedge s_1 < \perp] \\
\mathit{bank\_pay\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, n, \sigma) = 0 \\
[H = \mathit{Thief} \wedge y_{Bank,b} > n \wedge y_{Bank,to\_pay} \geq n \wedge s_1 > \perp] \\
\mathit{bank\_pay\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, n, \sigma) \\
= \mathit{runtime\_dispatch\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, 0, 0, 0, \sigma) \\
[H = \mathit{Thief} \wedge y_{Bank,b} \leq n] \\
\mathit{bank\_pay\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, n, \sigma) \\
= \mathit{runtime\_dispatch\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, 0, 0, 0, \sigma) \\
[H = \mathit{Thief} \wedge y_{Bank,to\_pay} \leq n] \\
\mathit{pay\_cont\_int\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, r, n, \sigma) = \mathit{runtime\_dispatch\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay} - n, y_{Thief,b}, 0, 0, 0, \sigma) \\
[\mathit{true}] \\
\mathit{thief\_fallback\_}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, \sigma) \\
= \mathit{bank\_pay\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, \mathit{Thief}, 0, 1, \sigma) \\
[H = \mathit{Bank}] \\
\mathit{runtime\_dispatch\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, r, \sigma) \\
= \mathit{pay\_cont\_int\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, s_4, s_3, r, s_5, \sigma{\uparrow}^6, \perp^6) \\
[s_1 = \mathit{Bank} \wedge s_2 = \mathit{pay\_cont\_int\_int}] \\
\mathit{runtime\_dispatch\_int}(x_{Bank,b}, x_{Bank,to\_pay}, x_{Thief,b}, y_{Bank,b}, y_{Bank,to\_pay}, y_{Thief,b}, H, v, r, \sigma) \\
= \mathtt{max}(0, x_{Bank,b} - y_{Bank,b})\,[s1 = \perp]
\end{gathered}$$

Output by `CoFloCo` on the first approximant when computing a bound to the amount of cryptocurrency lost is:

```
MAXIMUM LOSS:
### Maximum cost of
 main__(Bank__balance_,Bank_to_pay,Thief__balance_,
        _msg_sender_,_msg_value_,_ret_,N,
        S1,S2,S3,S4,S5,S6,S7,S8,S9,S10,S11,S12)
 : nat(N)
Asymptotic class: n
```

On the second approximant the output is:

```
MAXIMUM LOSS:
### Maximum cost of
 main__(Bank__balance_,Bank_to_pay,Thief__balance_,
        _msg_sender_,_msg_value_,_ret_,N,S1,S2,S3,S4,S5,
        S6,S7,S8,S9,S10,S11,S12,S13,S14,S15,S16,S17,S18)
 : nat(Bank__balance_-1)
Asymptotic class: n
```

The bound to the second approximant is not tight: it is easy to prove that the `Bank` can never lose more than $N+1$ coins when the computation uses at most two stack frames.

7.5. Alternative approach to CPS translations

As discussed in Ref. [9], it is possible to remove continuations (of imperative languages) without using any CPS translation. The key to avoiding CPS translations is to resort to a technique used in logic programming to encode imperative programming, since logic programming is the model of `CoFloCo`. We illustrate the feature with an example. Consider analyzing the cost of the function f defined in pseudo-C code as follows:

$$f(x)\{\ \texttt{return}(g(h(x)))\ \}\ .$$

One can use the cost equation

$$k_f(X,Z) = e + k_h(X,Y) + k_g(Y,Z) \qquad [\mathit{true}]$$

where each k_f, k_h, and k_g are the cost of the functions f, g, and h, respectively. In this equation, logic variables encode the return values of functions and, at the same time, the input of the continuation. In particular, Y, which encodes the return value of $h(x)$, is used as an output parameter of k_h and as an input parameter of k_g. The logical variable Z, which encodes the return value of f and g, is used as an output parameter of both k_f and k_g. By means of the above expedient, we could have avoided CPS-like translations and augmented our intermediate language with continuations to function calls. We decided not to do that for several reasons:

1. an intermediate language without continuations is much easier to be statically analyzed; in the future, we plan to try other techniques different from the generation of cost equations and the lack of continuations grants us more freedom in the choice of techniques;
2. Smart contract languages display failures with automatic backtracking and ad-hoc catch operations on errors. That is, these languages have explicit control operators. It turns out that CPS translation is the most flexible technique to encode languages with control operators (in languages without them). Adopting the CPS translation since the beginning allows easier scaling to more complex analyzes of future extensions of `mSCL`.

Finally, it is not clear a priori whether an alternative approach that avoids the CPS translation could scale better producing fewer equations. De Santis, in his Bachelor Thesis [12], has implemented the foregoing direct technique, which is entangled by the resolution of dynamic dispatch and the management of the initial memory to implement backtracking. After applying optimizations comparable to the ones in our prototype the result is that the two methods are incomparable in the number of equations generated (see Table 2, taken directly from Ref. [12]).

Table 2
Comparison of De Santis's direct translation to cost equations versus ours based on the CPS transformation.

Example	No CPS	CPS
	number of cost equations	number of cost equations
Bank-Thief code	11	20
English auction1	29	20
English auction2	27	38
Handover Ponzi	1438	336

7.6. Further extensions

The encoding of the frame stack for invocations given in Section 7.3 can be generalized to any array of bounded size[6]: if n is the maximum size of the array A, then A can be encoded by a sequence of n variables $a_1, \ldots, a_n$ plus a further variable *top* that records the current array length. Operations like array access, `push` and `pop` can then be implemented using large `if - then - else` decision trees. Actually, this pattern has been used to encode manually the Chain-Shaped Ponzi scheme in `mSCL`.

In a similar way, maps of finite domains, e.g. maps used to associate data with contract addresses, can be encoded as a set of pairs of variables, holding respectively a key and its associated value. At the moment, though, our prototype does not implement bounded arrays and finite maps.

7.7. Dynamic instantiation of smart contracts

Smart contract languages also feature the dynamic creation of contracts (cf. the operation `new` in Solidity). Our technique does not allow us to verify programs that use this operation in an unconstrained way, e.g. `new` inside an unbounded recursion or iteration. In all the other cases, the dynamic creation of smart contracts may be anticipated at static time by pre-instantiating the contract a finite number of times, thus paving the way to our analysis.

More precisely, as discussed in Section 6, we have analyzed smart contracts taken from `etherscan.io` and written in Solidity. We have found very few contracts that always use a statically bounded number of `new` (no contract uses the operation inside a recursion or an iteration). To test our analyzer, we rewrote the code by replacing the `new` with a set of pre-instantiated contract names. As expected, the overall result has been a blow-up of the equations because the set *Id* is augmented (see the foregoing Subsection 7.1).

7.8. Deployment

Our prototype is a static analyzer for `mSCL` that does not run any code: the successful codes will eventually run on a real blockchain. In De Santis's bachelor thesis [12], he has implemented a compiler from `mSCL` to Solidity. The compiler turns every `mSCL` contract into a Solidity contract with additional fields to hold the addresses of companion contracts. Moreover, the Solidity contracts have an additional function that, when called for the first time, receives the address of the companion contracts and stores them in the additional fields. Additionally, the compiler also returns a Python script that compiles and injects the Solidity code in Ethereum, creates a new contract instance for every `mSCL` contract and invokes each additional function to let each contract know the addresses of its companion contracts. Finally, the compiler performs type inference for translating function calls over addresses, since Solidity requires to cast every address to a contract interface.

[6] We are adopting the Solidity terminology: an array is a stack data structure that can grow dynamically, e.g. via `push` operations.

While the code in Ref. [12] is quite simple, it is already important from our perspective. Indeed, thanks to it, we can think of `mSCL` as a basic programming language for verifiable smart contracts that we can evolve in diverging directions from Solidity in order to strike a good balance between expressivity and the possibility of doing static analyses, in the spirit of other languages like Vyper [13], which even sacrifices Turing completeness for that.

8. Related works

In the past few years, formal methods have been largely used to analyze smart contracts to verify security properties. Our technique follows the same pattern of previous analyzers proposed in Refs. [14,15]. In those cases, the purpose of the analysis has been the over-approximation of the computational cost and the resource usage of actor-based programming languages.

A contribution that also addresses cryptocurrency movements in a subset of Solidity similar to `mSCL` is in Ref. [16]. They propose an analysis framework based on a compilation of the subset of Solidity to F^*, a functional language aimed at program verification with a powerful type and effect system. Using F^* types, it is possible to trace Ethers and discover critical patterns in smart contracts, such as reentrancy attacks. Unlike our technique, they are not able to derive the upper bounds of Ethers gained and lost by smart contracts.

A technique based on cost equations has been already applied to smart contract languages for analyzing gas-consumption [7]. In that work the authors analyze the Ethereum Virtual Machine code obtained from Solidity using classical control flow analysis where every node records the gas-consumption of the corresponding operation. The technique yields a precise analysis of conditional statements by restricting the language to guards belonging to Presburger arithmetic (similarly to what we do in this paper). There are similarities and differences between Ref. [7] and our paper. They use a cost analyzer to compute gas and use an intermediate language, which is called RBR. However, they address the bytecode instructions (for which gas is defined) and RBR is completely different from our intermediate language: it is imperative, uses memory locations, and abstracts over the instruction of a particular assembly language. In this paper, we are interested in a property that is expressed on the high level code, where the programmer has a better grasp of the invariants. For this reason, our intermediate code abstracts away from the instructions of a high level language. The important difference between this paper and Ref. [7] is the following one. Computing gas amounts to over-approximate a function $GAS(x)$ (see our Definition 7) and, in Ref. [7], the authors define this over-approximation by abstracting out from the identity of smart contract addresses. This abstraction is not possible when one has to compute balances because confusing one smart contract address with another may lead to awful errors. It is exactly this analysis of smart contract addresses that causes the huge number of cost equations, even exponential with respect to the input, which is not the case in Ref. [7]. Finally, up-to our understanding, Albert et al. analyze one smart contract at a time, and, looking at the examples in the paper and the ones pre-loaded in the on-line prototype, that smart contract never calls methods of other smart contracts. Instead, our analyzer verifies sets of interacting smart contracts and, in particular, the cases of reentrant codes.

An interesting paper about asset movements targets Bitcoin Script [6]. In that work, the authors verify the absence of assets that remain frozen in contracts, i.e. liquidity. In particular, they prove the decidability of liquidity in a model of Bitcoin Script, called BitML. We think that our technique is adequate to reason about liquidity as well, and it would be interesting to compare the two approaches on `mSCL`.

As regards intermediate languages, other languages have been defined for smart contract analysis (apart from RBR mentioned above). One such language is Scilla [17] that is based on communicating automata that are stateful and use updates. At the moment, the model of Scilla does not feature exceptions and, therefore, it is not clear how to model rollbacks. Vandal, defined in Ref. [18], converts Ethereum Virtual Machine bytecode to semantic logic relations. These relations, paired with the security analysis expressed as logic rules, produce outputs listing potential vulnerabilities. Also, Vandal does not model backtrack: it reduces to flagging "vulnerable" all those actions that may cause rollback. So, as far as we can see, we could have used neither Scilla nor Vandal to define `mSCL` behaviors.

Other formal techniques have addressed the critical interplay between smart contracts and users (that are usually untrusted) [19-22]. In these cases, the model is nondeterministic (because of users' behavior) and one tries to predict the maximum profit for some users. The proposed techniques range from game theory to symbolic analysis of computations and to (decidable fragments of) temporal logic. In this paper, we focus on (deterministic) behaviors and compute the best and the worst possible scenarios of smart contract compositions. That is, if we want to analyze the interaction with a possible user, we need to express the user as a deterministic contract.

9. Conclusions

In this paper, we have analyzed cryptocurrency movements of smart contracts written in a lightweight version of Solidity, called `mSCL`, which is procedural and features dynamic dispatch. The analysis yields cost equations defining upper bounds of loss and gain of smart contracts that are computed by means of an off-the-shelf cost analyzer. The definition of the cost equations has been given by means of a simple functional language with static dispatch that expresses the input-output behavior of `mSCL` functions. Our technique has been prototyped and we have reported its assessments and discussed extensions with additional features to partially cover the gap with mainstream smart contract languages.

Several extensions of the analyzer need to be investigated in the near future. They mostly concern the management of other data types and other operations, such as modifiers, try-catch instructions, etc.

Another important research direction concerns the study of optimizations for our prototype. The encoding of the extended language with the `address` data type gives a number of cost equations that are exponential with respect to the `address` variables in the source `mSCL` code. This explosion is due to having chosen a simple and intelligible encoding. For example, if one encodes a function *f* that just passes the `msg.sender` to another function *g*, the resulting equations for *f* and *g* will have two disjoint sums over all possible addresses, while one sum would have been sufficient (and it might be the case that no sum is necessary at all, e.g. the `msg.sender` is never used anyway in *g*). To avoid this pitfall, our current prototype (which admits `address` types) already refines the encoding by using optimizations that greatly reduce the size of the output. These optimizations are not very aggressive at the moment: they only remove conditionals with identical branches and merge identical alternatives in choices. Table 3 reports the results of these optimizations for the Chain-shaped Ponzi scheme in Ref. [11] when the smart contracts involved are 2, 3, and 4: the reader may notice that the number of equations decreases by 87% on average. A formal study of (more aggressive) optimizations has not been undertaken so far and is on our agenda.

Our analysis is symbolic and fully automatic, which are evident pros. Two cons of the technique are: (i) it is not clear how much approximated it is, namely how much tight are the maximal loss and maximal gain we compute, and, up-to optimizations, (ii) the number of equations we produce are exponential in the code size because cost solvers are too

Table 3
Results of current optimizations for the Chain-shaped Ponzi scheme.

♯ Smart Contracts	♯ Equations without optimizations	♯ Equations with optimizations
2	6,492	1,030
3	98,133	11,825
4	1,452,566	170,308

rigid. However, there are other analyses and techniques one can try and that we intend to investigate. In particular, one may benefit from the simplicity of our intermediate language that, being functional, first-order and with static dispatch, is a simple target for formal methods. Therefore, one could trade automation for precision and scalability, manually proving tight bounds by means of interactive provers, like Refs. [23,24], using functional languages with expressive types systems, like `F*` [25], or even combining them with amortized analysis in the spirit of Ref. [26] (in the Chain-shaped Ponzi scheme a potential can easily be attached to the queue of current participants to compute the maximal gain). These approaches also allow one to perform the analysis of functional and non-functional properties at once.

Finally, while gas consumption has been overlooked in this paper, its analysis is relevant and we are going to address it. Indeed, gas consumption decreases the maximal gain and increases the maximal loss, thus triggering unwanted backtracking in case of lack of gas. Previous work on gas focuses on the direct analysis of the bytecode, whereas we work directly on the source code. However, the two approaches can be reconciled using the technique defined in the project CerCo [27]: an instrumented compiler can produce at once the bytecode and it is possible to define a precise cost model for the source language where the cost of every basic block is induced by the cost of the bytecode that corresponds to that block.

Declaration of competing interest

The authors declare that they have no known competing financialinterests or personal relationships that could have appearedto influence the work reported in this paper.

Fundings

Cosimo Laneve has been partly supported by the H2020-MSCA-RISE project ID 778233 'Behavioural Application Progrvdam Interfaces (BEHAPI)'. Claudio Sacerdoti Coen has been partially supported by MIUR-PRIN project 'Analysis of Program Analyses' (ASPRA, ID 201784YSZ5_004) and the Italian INdAM—GNCS project 2020 'Reversible Concurrent Systems: from Models to Languages'.

Appendix A. Technical details

Lemma 16. (***Substitution Lemma***) *Let Γ_0, Γ_1 be pure environments and Γ, Γ' be ground environments. If* $\Gamma_0,\Gamma_1[\overline{x}\mapsto\overline{x_0},\overline{y}\mapsto\overline{0}]\vdash^{z}_{C,D}\mathsf{S}:\Theta$ *then* $\Gamma,\Gamma'[\overline{x}\mapsto\overline{v},\overline{y}\mapsto\overline{0}]\vdash^{u}_{C,D}\mathsf{S}:\Theta\{u,\overline{v}/z,\overline{x_0}\}[\Gamma_0,\Gamma_1\rightsquigarrow\Gamma,\Gamma']$. *Similarly for expressions.*

Proof. Standard induction on the depth of the proof tree of $\Gamma_0,\Gamma_1[\overline{x}\mapsto\overline{x_0},\overline{y}\mapsto\overline{0}]\vdash^{z}_{C,D}\mathsf{S}:\Theta$ and a case analysis on the last rule used.

Theorem 5. Let $\mathcal{P}$ be a `mSCL` program such that $\vdash\mathcal{P}:\mathcal{I}$ and let $\mathcal{S}$ be an initial state such that $\mathcal{S}\vdash\Theta$. Then

1. (*determinism*) If $\Theta[xrArr]^{*}_{\mathcal{I}}\Theta'$ then there is at most one Θ'' such that $\Theta'[xrArr]_{\mathcal{I}}\Theta''$;
2. (*correctness*) If $\mathcal{S}\longrightarrow\mathcal{S}'$ then there exists Θ' such that $\mathcal{S}'\vdash\Theta'$ and $\Theta[xrArr]^{*}_{\mathcal{I}}\Theta'$.

Proof. Determinism. This follows directly from the translation in Section 4 because, in every $\sum_{i\in1..n}(\varphi_i)\ \Theta_i$, at most one φ_i may be true every time.

Correctness. The proof is by induction on the length of $\mathcal{S}\longrightarrow\mathcal{S}'$. We use the property that, if $\Gamma\vdash^{v}_{C,D}e:e'$ and $[\![e]\!]_{C,v,D,\ell}=v'$ where ℓ is the memory of D in some state $\mathcal{S}''$ then e' is a ground expression whose value $[\![e']\!]$ is v'.

The basic case of the induction is immediate; the inductive case

$$\mathcal{S}\longrightarrow\mathcal{S}'\longrightarrow\mathcal{S}''$$

is demonstrated by means of a case-analysis on the reduction $\mathcal{S}'\longrightarrow\mathcal{S}''$.

We discuss only the sub-case when $\mathcal{S}'\longrightarrow\mathcal{S}''$ uses rule [METH]; the other ones are either simpler or similar. Since we are using [METH], the following are true:

$$\mathcal{S}'=\prod_{i\in1..n}C_i(\ell'_i\cdot\ell_i)\,|\,C_j\overset{v}{\blacktriangleright}C_k:e.\mathtt{m}(\overline{e'})\tag{1}$$

$$[\![e]\!]_{C_j,v,C_k,\ell'_k}=C_h\tag{2}$$

$$[\![\overline{e'}]\!]_{C_j,v,C_k,\ell'_k}=\overline{v'}\tag{3}$$

$$\mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ y};\mathsf{S}_{\mathtt{m}}\}\in\mathsf{C}_h\tag{4}$$

$$\Gamma,\Gamma'=envs(\mathcal{S}')\tag{5}$$

Additionally, since $\mathcal{S}'\vdash\Theta'$ for some Θ' by induction hypothesis, we have used rule [INVK-NV] with the hypotheses:

$$\Gamma'\vdash^{v}_{C_j,C_k}e:e_0\tag{6}$$

$$\Gamma'\vdash^{v}_{C_j,C_k}\overline{e'}:\overline{e''}\tag{7}$$

By definition of $envs(\mathcal{S}')$ we have $\Gamma'(C_k)=\ell_k'$; therefore, by (2) and (6) we have $e_0=C_h$ and, by (3) and (7), $\overline{e''}$ are ground expressions whose values are $\overline{v'}$.

According to rule [METH], we have

$$\mathcal{S}''=\prod_{i\in(1..n)\setminus h}C_i(\ell'_i\cdot\ell_i)\,|\,C_h(\ell'_h[\overline{x}\mapsto\overline{v'},\overline{y}\mapsto\overline{0}],\ell_h)\,|\,C_k\overset{0}{\blacktriangleright}C_h:\mathsf{S}_{\mathtt{m}}$$

where $\mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ y};\ \mathsf{S}_{\mathtt{m}}\} \in C_h$. We demonstrate that there is Θ'' such that $\mathcal{S}'' \vdash \Theta''$ and $\Theta' \Rightarrow_{\mathcal{I}} \Theta''$. By rule [FUNCTION],

$$\Gamma_0, \Gamma_1[\overline{x} \mapsto \overline{x_0}, \overline{y} \mapsto \overline{0}] \vdash^{0}_{C_k,C_h} \mathsf{S}_{\mathtt{m}} : \Theta_{C_k}$$

and, by the Substitution Lemma, we obtain

$$\Gamma, \Gamma'[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{0}] \vdash^{0}_{C_k,C_h} \mathsf{S}_{\mathtt{m}} : \Theta_{C_k}\{\overline{v}/\overline{x_0}\}[\Gamma_0, \Gamma_1 \rightsquigarrow \Gamma, \Gamma'] \ . \qquad (8)$$

By definition (8) is exactly $\mathcal{S}'' \vdash \Theta_{C_k}\{\overline{v}/\overline{x_0}\}[\Gamma_0, \Gamma_1 \rightsquigarrow \Gamma, \Gamma']$.
As regards Θ', we observe that

$$\begin{array}{rl} \Theta' & = (m \in C_h)\ C_h.m(\Gamma, \Gamma', 0, \overline{v'}, C_k) \\ & +\ \ (m.payable \in C_h)\ C_h.m(\Gamma, \Gamma', 0, \overline{v'}, C_k) \\ & +\ \ (m \notin C_h\ \wedge\ m.payable \notin C_h\ \wedge\ fallback \in C_h)\ \Gamma' \\ & +\ \ (m \notin C_h\ \wedge\ m.payable \notin C_h\ \wedge\ fallback \notin C_h)\ \Gamma \end{array}$$

and, by (4), the unique valid alternative in Θ' is the first one. Therefore the evaluation of Θ' amounts to unfold the function invocation $C_h.m(\Gamma, \Gamma', 0, \overline{v'}, C_k)$, that is

$$\Theta' \Rightarrow_{\mathcal{I}} \Rightarrow_{\mathcal{I}} \Theta_{C_k}\{\overline{v}/\overline{z}\}[\Gamma_0, \Gamma_1 \rightsquigarrow \Gamma, \Gamma']$$

This concludes the proof.

Theorem 12. (*Correctness of cost equation generation*). *Let $\mathcal{P}$ be an* mSCL *program, $\mathcal{S}$ be an initial state and $\vdash \mathcal{P} : \mathcal{I}$ and $\mathcal{S} \vdash C.m(\Gamma, \Gamma, v', \overline{v}, H)$ and $C.m(\Gamma, \Gamma, v', \overline{v}, H)[xrArr]^{*}_{\mathcal{I}}\Gamma'$. Let us extend $\langle \ulcorner \mathcal{I} \urcorner \rangle$ (where we use either $cost^{C'}_{gain}$ or $cost^{C'}_{loss}$ during the translation) with a main function that calls C.m. Then*

1. *Determinism: $main(\lfloor\Gamma\rfloor, v', \overline{v}, H)$ has a unique $\rightarrow_{\mathtt{CoFloCo}}$-normal-form*
2. *Correctness:*
   - *$main(\lfloor\Gamma\rfloor, v', \overline{v}, H) \rightarrow_{\mathtt{CoFloCo}}{}^{*} \mathrm{GAIN}^{C'}_{\mathcal{I},C,m}(\Gamma, v', \overline{v}, H)$ if we selected $cost^{C'}_{gain}$ during the translation,*
   - *$main(\lfloor\Gamma\rfloor, v', \overline{v}, H) \rightarrow_{\mathtt{CoFloCo}}{}^{*} \mathrm{LOSS}^{C'}_{\mathcal{I},C,m}(\Gamma, v', \overline{v}, H)$ if we selected $cost^{C'}_{loss}$ during the translation.*

Proof. (*Sketch*) The proof is by accumulation of intermediate facts:

1. for every ground Presburger expression e whose value is v, $\ulcorner e \urcorner \rightarrow_{\mathtt{CoFloCo}} v$.

   Proof: by inspection of the definition of $\ulcorner e \urcorner$.

2. for every ground guard φ, φ holds if and only if $\ulcorner \varphi \urcorner$ holds.

   Proof: by inspection of the definition of $\ulcorner \varphi \urcorner$, remembering that $\ulcorner \cdot \urcorner$ is injective over functions and contract names.

3. for every code Θ_1, *if* $\Theta_1 \Rightarrow_{\mathcal{I}} \Theta_2$ in the intermediate language and $\ulcorner \Theta_1 \urcorner = \Theta'_1$ *and* $\ulcorner \Theta_2 \urcorner = \Theta'_2$, *then* $\Theta'_1[xrArr]_{\mathcal{I}}\Theta'_2$ in the simplified intermediate language.

Proof: Here we are abusing of the notation because the terms Θ'_1 *and* Θ'_2 are not in the intermediate language but are in the simplified one (which does not use environments at all). Similarly when we write $\Theta'_1[xrArr]_{\mathcal{I}}\Theta'_2$. However, the simplified language syntax and semantics are almost identical to those of the intermediate language but for:

(a) functions take in input only lists of variables (in the intermediate language the first two arguments were environments);
(b) the final code is a list of values instead of an environment.

As regards the semantics, the rules of the simplified intermediate language are exactly the same of the intermediate one. Said this, the proof is by inspection of the definition of $\ulcorner \Theta \urcorner$ over codes, using the previous two points.

4. let $m(\lfloor\Gamma_0\rfloor, \overline{x}) = \sum_{i\in I}\left(\bigvee_{j\in J^i}\varphi^j_i\right)(\Theta_i)$ in the simplified intermediate language. Then, for every ground $\Gamma'_0, \Gamma, \overline{e}$ *and n such that* $m(\lfloor\Gamma'_0\rfloor, \overline{e})[xrArr]^{n}_{\mathcal{I}}\lfloor\Gamma\rfloor$ with exactly one derivation, there is exactly one derivation $m(\lfloor\Gamma'_0\rfloor, \overline{e}) \rightarrow_{\mathtt{CoFloCo}}{}^{*} cost(\lfloor\Gamma'_0\rfloor, \lfloor\Gamma\rfloor)$ (the derivation is deterministic).

Proof: by induction on n. *Let* $\bigvee_{j\in J^i}\varphi^j_i$ be the unique guard such that $(\bigvee_{j\in J^i}\varphi^j_i)\{\lfloor\Gamma'_0\rfloor / \lfloor\Gamma_0\rfloor\}\{\overline{e}/\overline{x}\}$ holds. This can happen only if there is at least one $j \in J^i$ such that $\varphi^j_i\{\lfloor\Gamma'_0\rfloor / \lfloor\Gamma_0\rfloor\}\{\overline{e}/\overline{x}\}$ holds. Then

$$m(\lfloor\Gamma'_0\rfloor, \overline{e})[xrArr]_{\mathcal{I}}\Theta_i\{\lfloor\Gamma'_0\rfloor / \lfloor\Gamma_0\rfloor\}\{\overline{e}/\overline{x}\}$$

and

$m(\lfloor\Gamma_0^{'}\rfloor,\bar{e}) \rightarrow_{\texttt{CoFloCo}} cost(\lfloor\Gamma_0^{'}\rfloor,\Theta_i\{\lfloor\Gamma_0^{'}\rfloor / \lfloor\Gamma_0\rfloor\}\{\bar{e} / \bar{x}\})$

because of the equation

$m(\lfloor\Gamma_0\rfloor,\bar{x}) = cost(\lfloor\Gamma_0\rfloor,\Theta_i) \qquad [\varphi_i^j]$

No other $\rightarrow_{\texttt{CoFloCo}}$-reduction is possible because:

(a) *any other predicate $\varphi_{i'}^j\{\lfloor\Gamma_0^{'}\rfloor / \lfloor\Gamma_0\rfloor\}\{\bar{e} / \bar{x}\}$ for $i' \neq i$ and $j \in J^{i'}$ evaluates to false. This follows by the fact that every other $(\bigvee_{j\in J^{i'}}\varphi_{i'})\{\lfloor\Gamma_0^{'}\rfloor / \lfloor\Gamma_0\rfloor\}\{\bar{e} / \bar{x}\}$ must be false, otherwise the reduction should not have been deterministic on the intermediate terms (obtained translating mSCL programs);*

(b) *any other disjunct $\varphi_i^{j'}\{\lfloor\Gamma_0^{'}\rfloor / \lfloor\Gamma_0\rfloor\}\{\bar{e} / \bar{x}\}$ of $\bigvee_{j\in J^i}\varphi_i^j\{\lfloor\Gamma_0^{'}\rfloor / \lfloor\Gamma_0\rfloor\}\{\bar{e} / \bar{x}\}$ that evaluates to true triggers the same reduction*

$m(\lfloor\Gamma_0^{'}\rfloor,\bar{e}) \rightarrow_{\texttt{CoFloCo}} cost(\lfloor\Gamma_0^{'}\rfloor,\Theta_i\{\lfloor\Gamma_0^{'}\rfloor / \lfloor\Gamma_0\rfloor\}\{\bar{e} / \bar{x}\})$

because of the equation

$m(\lfloor\Gamma_0\rfloor,\bar{x}) = cost(\lfloor\Gamma_0\rfloor,\Theta_i) \qquad [\varphi_i^{j'}]$

By cases over n:

- *if $n = 0$ then $\Theta_i\{\lfloor\Gamma_0^{'}\rfloor / \lfloor\Gamma_0\rfloor\}\{\bar{e} / \bar{x}\} = \lfloor\Gamma\rfloor$ and we are done;*
- *otherwise $\Theta_i\{\lfloor\Gamma_0^{'}\rfloor / \lfloor\Gamma_0\rfloor\}\{\bar{e} / \bar{x}\} = m^{'}(\lfloor\Gamma_0\rfloor,\overline{e^{'}})[xrArr]_{\mathcal{I}}^{n-1}\lfloor\Gamma\rfloor$ and $cost(\lfloor\Gamma_0\rfloor, m^{'}(\lfloor\Gamma_0\rfloor,\overline{e^{'}})) = m^{'}(\lfloor\Gamma_0\rfloor,\overline{e^{'}})$ and we conclude by inductive hypothesis.*

5. *Determinism: $main(\ulcorner\Gamma\urcorner,\ulcorner\Gamma\urcorner,e,\bar{x},\ulcorner H\urcorner)$ has a unique $\rightarrow_{\texttt{CoFloCo}}$-normal-form as a corollary of all previous points and since the new equation main has only one branch*
6. *Correctness:*
   - *$main(\lfloor\Gamma\rfloor,v^{'},\bar{v},H) \rightarrow_{\texttt{CoFloCo}}{}^{*} \mathrm{GAIN}_{\mathcal{I},C,m}^{C^{'}}(\Gamma,v^{'},\bar{v},H)$ if we selected $cost_{gain}^{C^{'}}$ during the translation,*
   - *$main(\lfloor\Gamma\rfloor,v^{'},\bar{v},H) \rightarrow_{\texttt{CoFloCo}}{}^{*} \mathrm{LOSS}_{\mathcal{I},C,m}^{C^{'}}(\Gamma,v^{'},\bar{v},H)$ if we selected $cost_{loss}^{C^{'}}$ during the translation.*

By the previous point, both $\mathrm{GAIN}_{\mathcal{I},C,m}^{C^{'}}$ *and* $\mathrm{LOSS}_{\mathcal{I},C,m}^{C^{'}}$ are defined. Combining the first 5 points, we obtain $main(\ulcorner\Gamma\urcorner,\ulcorner\Gamma\urcorner,e,\bar{x},\ulcorner H\urcorner) \rightarrow_{\texttt{CoFloCo}}{}^{*} cost(\ulcorner\Gamma\urcorner, \ulcorner\Gamma^{'}\urcorner)$ *where*

$\mathtt{m}(\Gamma,\Gamma,e,\bar{x},H)[xrArr]_{\mathcal{I}}^{*}\Gamma^{'}$. *Therefore* $\mathrm{GAIN}_{\mathcal{I},C,m}^{C^{'}}(\Gamma,v^{'},\bar{v},H) = \mathtt{max}(0,\Gamma^{'}(C^{'}.balance) - \Gamma(C^{'}.balance))$, $\mathrm{LOSS}_{\mathcal{I},C,m}^{C^{'}}(\Gamma,v^{'},\bar{v},H) = \mathtt{max}(0,\Gamma(C^{'}.balance) - \Gamma^{'}(C^{'}.balance))$,

$cost_{gain}^{C^{'}}(\ulcorner\Gamma\urcorner,\ulcorner\Gamma^{'}\urcorner) = \mathtt{max}(0,\Gamma^{'}(C^{'}.balance) - \Gamma(C^{'}.balance))$ *and*

$cost_{loss}^{C^{'}}(\ulcorner\Gamma\urcorner,\ulcorner\Gamma^{'}\urcorner) = \mathtt{max}(0,\Gamma(C^{'}.balance) - \Gamma^{'}(C^{'}.balance))$.

The thesis holds.

References

[1] Harris Brakmić, cript, in: Bitcoin S, Apress, Berkeley, CA, 2019, pp. 201–224.
[2] Chris Dannen, Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners, Apress, Berkely, USA, 2017.
[3] Sam Blackshear, et al., Move: a language with programmable resources, Available at, https://developers.libra.org/docs/assets/papers/libra-move-a-language-with-programmable-resources.pdf, 2019.
[4] David Siegel, Understanding the Dao Attack. Retrieved June, 13:2018, 2016.
[5] Lorenz Breidenbach, Phil Daian, Ari Juels, Emin Gun Sirer, An in-depth look at the parity multisig bug, Available at, http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/, 2017.
[6] Massimo Bartoletti, Roberto Zunino, Verifying liquidity of bitcoin contracts, in: Flemming Nielson, David Sands (Eds.), Principles of Security and Trust, Springer International Publishing, 2019, pp. 222–247.
[7] Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, Albert Rubio, SAFEVM: a safety verifier for ethereum smart contracts, in: Proc. of Software Testing and Analysis, ISSTA'19, ACM, 2019, pp. 386–389.
[8] Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, Martin Vechev, Securify: practical security analysis of smart contracts, in: Proc. of Computer and Communications Security, CCS '18, ACM, 2018, pp. 67–82.
[9] Antonio Flores Montoya, Reiner Hähnle, Resource analysis of complex programs with cost equations, in: Proceedings of 12th Asian Symposium on Programming Languages and Systems, Volume 8858 of Lecture Notes in Computer Science, Springer, 2014, pp. 275–295.
[10] Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, Closed-form upper bounds in static cost analysis, J. Autom. Reas. 46 (2) (Feb 2011) 161–203.
[11] Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, Roberto Saia, Dissecting Ponzi schemes on ethereum: identification, analysis, and impact, Future Generat. Comput. Syst. 102 (2020) 259–277.
[12] Stefano De Santis, Compilazione, deployment e analisi statica di un linguaggio per distributed applications, Bachelor Thesis, University of Bologna, 2020.
[13] Mudabbir Kaleem, Anastasia Mavridou, Aron Laszka, Vyper: a security comparison with solidity based on common vulnerabilities, BRAINS (2020) 107–111. IEEE, 2020.
[14] Abel Garcia, Cosimo Laneve, Michael Lienhardt, Static analysis of cloud elasticity, Sci. Comput. Program. 147 (2017) 27–53.
[15] Cosimo Laneve, Michael Lienhardt, Ka I. Pun, Guillermo Román-Díez, Time analysis of actor programs, J. Log. Algebr. Meth. Program. 105 (1–27) (2019).
[16] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al., Formal verification of smart contracts: short paper, in: Proc. of Programming Languages and Analysis for Security, ACM, 2016, pp. 91–96.
[17] Ilya Sergey, Amrit Kumar, Aquinas Hobor, Scilla: A Smart Contract Intermediate-Level Language, 2018, 00687. CoRR, abs/1801.
[18] Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A Scalable Security Analysis Framework for Smart Contracts. CoRR, abs/1809.03981, 2018.
[19] Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, Emilio Tuosto, Validation of decentralised smart contracts through game theory and formal methods, in: Programming Languages with Applications to Biology and Security, Volume 9465 of Lecture Notes in Computer Science, Springer, 2015, pp. 142–161.
[20] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, Aquinas Hobor, Making smart contracts smarter, in: Proc. of the Conference on Computer and Communications Security, ACM, 2016, pp. 254–269.
[21] Bernhard Mueller, Smashing Ethereum Smart Contracts for Fun and Real Profit, HITB SECCONF Amsterdam, 2018.
[22] Cosimo Laneve, Claudio Sacerdoti Coen, Adele Veschetti, On the prediction of smart contracts' behaviours, in: From Software Engineering to Formal Methods and Tools,

and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday, Volume 11865 of Lecture Notes in Computer Science, Springer, 2019, pp. 397–415.

[23] Benjamin C. Pierce, Arthur Azevedo de Amorim, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hriţcu, Vilhelm Sjöberg, Andrew Tolmach, and Brent Yorgey. Programming language foundations, Software Found. Ser. 2 (May 2018). Electronic textbook.

[24] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, Matita tutorial, J. Formaliz. Reason. 7 (2) (2014) 91–199.

[25] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, Santiago Zanella-Béguelin, Dependent types and multi-monadic effects in F*, in: 43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), ACM, January 2016, pp. 256–270.

[26] Hofmann Martin, Automatic amortized analysis. In Moreno Falaschi and Elvira Albert, in: Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015, ACM, 2015, p. 5.

[27] M. Roberto, Amadio, Nicholas Ayache, François Bobot, Jaap Boender, Brian Campbell, Ilias Garnier, Antoine Madet, James McKinna, Dominic P. Mulligan, Mauro Piccolo, Randy Pollack, Yann Régis-Gianas, Claudio Sacerdoti Coen, Ian Stark, Paolo Tranquilli, Certified complexity (cerco), in: Ugo Dal Lago, Ricardo Peña (Eds.), Foundational and Practical Aspects of Resource Analysis - Third International Workshop, FOPARA 2013, Bertinoro, Italy, August 29-31, 2013, Revised Selected Papers, Volume 8552 of Lecture Notes in Computer Science, Springer, 2013, pp. 1–18.