Analysis of smart contracts balances

Cosimo Laneve¹, Claudio Sacerdoti Coen*,²

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

ABSTRACT

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 behaviours 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 [10], Solidity for Ethereum [13], Move for Libra [9]. 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 [11,25].

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 [2], and compliance and violation of programming patterns [27]. 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 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 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 behaviour, 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 \[1,14]. These cost analyzers use (declarative) languages that have a rigid structure and a 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 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 arithmetic. Functions in the intermediate language are intended to define the input-output behaviour 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 has required an explicit encoding that entangle 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 an higher-order language for the Continuing 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 to 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 to notice 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 few archetypical 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 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 \[25]. 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 compile 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 \[1,2].

### 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 behaviour, cryptocurrency transfer, recursion and failures that are inspired to Solidity.

We use a countable set of smart contract names \(C\), ranged over by \(C, D, H\), a countable set of function names, ranged over by \(f, \ell\), a countable set of field names \(F_2\), ranged over by \(f, \ell\), and a countable set \(V\) of variables, ranged over by \(x, y, z\). Variables include field names and smart contract names.

The syntax of mSCL is

\[
\begin{align*}
C & ::= \text{contract } C \{ \quad \mathbf{T} \quad ; \quad F \quad [\text{fallback()} \quad \text{payable} \{\} ] \} \\
F & ::= \epsilon \mid \text{function } m(Tx) \quad [\text{payable} \{\} \quad \mathbf{T} \quad ; \quad S \} \quad F \\
T & ::= \text{uint} \\
S & ::= \epsilon \mid x = E ; \quad \text{if } (E) \quad \{ S \} \quad \text{else } \{ S \} \mid \text{E.m.value}(E)(E) ; \quad \text{revert} ; \quad \text{E.transfer}(E) ; \\
SE & ::= n \mid x \mid \text{this} \mid E \# E \mid E \mid \text{msg.sender} \mid \text{msg.value} \mid \text{E.balance} \\
\# & ::= + \mid - \mid > \mid = \mid \geq \mid && \text{a} \& \ast \mid / \\
\end{align*}
\]

where terms written within “[“ and “]” are optional. A mSCL program \(P\) is a sequence of smart contract definitions \((C_1, \ldots, C_n)\), which, in turn, are sequences of fields and function definitions. If \(C = \text{contract } C \{ \ldots \} \), we say that \(C\) is the smart contract name of \(C\) and we address the set of contract names of \(P\) with \(\text{contractnames}(P)\).

In a contract \(\text{contract } C \{ \quad \mathbf{T} \quad ; \quad F \quad [\text{fallback()} \quad \text{payable} \{\} ] \} \), the fields are \(\mathbf{T}\); and the corresponding set is \(\text{fields}(C)\), the functions are either
those in \( F \) or the fallback. We write \( m(\mathbb{T}x)[\text{payable}][\mathbb{T}x; S] \in \mathcal{C} \) if the function belongs to the contract named \( C \) and similarly for fallback \( C \). Additionally, in function \( m \) function \( m(\mathbb{T}x)[\text{payable}][\mathbb{T}x; S], \mathbb{T}x \) are the formal parameters and \( \mathbb{T}x; S \) is the body of \( m \), where \( \mathbb{T}x \) 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 \( \epsilon \), the assignment \( x = E \) 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 \( E.m(E) \) and \( E.mvalue(E)(E) \), where \( E \) is the call context, \( m \) is the function and \( E \) are the actual parameters; the term value \( E^{\prime} \) 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).

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 a 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 to notice 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 \( \text{uint} \) and \( \text{address} \). Local variables and function parameters are typed by \( \text{uint} \) and the only expressions typed by \( \text{address} \) are \( \text{msg.sender}, \text{this} \) and the names of the smart contracts defined in the program. In particular the type systems ensures 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 \( \text{address} \).

Our technique will admit external callers.

**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 \( \text{pay} \) 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 to draw 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 \( \text{pay} \) 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 \( \text{pay} \) also acknowledges the withdrawal by invoking client’s function \( \text{ack} \) (because the client has payed 1 cryptocurrency for it) – line 6. This apparently harmless operation is at the core of the attack because the \( \text{ack} \) function of Thief calls \( \text{pay} \) and the process continues till the account is emptied (e.g. the boolean expression at line 4 becomes false). The invocation \( \text{Bank.pay.value}(1)(2) \) 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 modelled in mSCL. First, nonempty fallbacks. 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 allow 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 \( \ell, \ell’, \ldots \), which are maps \( \text{Idl} \cup \text{Var} \rightarrow \mathbb{N} \). The following auxiliary functions are used in the semantic rules:

**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.**
is the memory update, namely $\text{let } f = f'\text{ in } v$ and $\text{let } f = f'\text{ in } v'$.

\[
\begin{align*}
\text{[if-true]} & : C' \circ C : e \leftarrow f \\
\text{[if-false]} & : C' \circ C : e \leftarrow f' \\
\text{[true]} & : C' \cdot C : x = \epsilon; S \rightarrow C'(x \rightarrow v') \cdot C' \cdot C : S
\end{align*}
\]

\[
\begin{align*}
\text{[false]} & : C' \cdot C : x = \epsilon; S \rightarrow C'(x \rightarrow v') \cdot C' \cdot C : S
\end{align*}
\]

\[
\begin{align*}
[\text{e}] C' \cdot C : \neg C' \cdot C : \top
\end{align*}
\]

\[
\begin{align*}
[\text{e}] C' \cdot C : \neg C' \cdot C : \top
\end{align*}
\]

\[
\begin{align*}
\text{State transitions } \rightarrow^\mu \text{ of mSCL} & = \rightarrow \cup \rightarrow^\text{fail}, \quad [\text{e}] \not\in \rightarrow^\mu \text{ never fails.}
\end{align*}
\]

- $\text{let } f \rightarrow v \text{ is the memory update, namely } (\ell[f \rightarrow v]) = v$ and $\ell(f \rightarrow v) = g$.
- $[\text{e}] C_{x,C} : e' \cdot e$ 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 $C$ be the memory of $C$ where values of fields and variables occurred in $e$ are stored. We omit the Definition of $[\text{e}] C_{x,C} : e' \cdot e$. 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.

\[
\begin{align*}
\text{A state of a mSCL program } P, \text{ ranged over by } S, S', \ldots, \text{ is defined by the following syntax}
\end{align*}
\]

\[
\begin{align*}
S ::= \prod_{i \in I} C_i \cdot (e_i', e_i) | C' \cdot C : S | \prod_{i \in I} C_i \cdot (e_i', e_i) | 0
\end{align*}
\]

where $\prod_{i \in I} C_i \cdot (e_i', e_i)$ is a parallel composition of (runtime) contracts and either $C' \cdot C : S$ 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 $\ell$ is the current memory and $\ell'$ is the backtrack memory. The memory $\ell'$ is the one at the beginning of the current transaction; $\ell$ is a working copy of $\ell'$, which is updated during the transaction and it is committed if the transaction ends successfully, becoming the new backtrack memory. When we write $C(\ell \cdot \ell')$, we always assume that $\text{dom}(\ell) = \text{fields}(C) \subseteq \text{dom}(\ell')$ (because $\ell$ also defines formal parameters and local variables). We say that a state is final when the runtime statement is of the form $\prod_{i \in I} C_i \cdot (e_i', e_i)$. 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 $P$.

Runtime statements may be either 0, the terminated statement, or $C(\ell \cdot \ell')$, where $C$ must be evaluated into the contract $C$, with a caller $C_0$ and with a value $v$.

The semantics of mSCL programs is defined by means of the transition

\[
\begin{align*}
S \rightarrow S' & \quad \text{[exit]} \quad S \rightarrow S' \quad \text{[fail]} \quad S \rightarrow S'
\end{align*}
\]

\[
\begin{align*}
F. 2. \text{ State transitions } \rightarrow^\mu \text{ of mSCL}
\end{align*}
\]
relation $S \xrightarrow{\delta_{\mathit{S}}} S'$, where $\delta_{\mathit{S}} = \delta \cup \delta_{\mathit{S}}(\text{trans})$ (the program is kept implicit in the notation). In an $\xrightarrow{\cdot}$-derivation to a final state, all transitions are $\xrightarrow{\cdot}$-except the last one that is responsible for committing the memory. In particular, if the last transition is $\xrightarrow{\cdot}$, then the computation terminates normally and the current memory becomes the new initial memory; if the last transition is $\xrightarrow{\cdot}$, then the computation backtracks and the current memory is reverted to the initial memory. The formal Definition of $\xrightarrow{\cdot}$ is given in Fig. 2.

Let us comment 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 [TRANS] and [TRANSFER-FAIL] define the semantics of $e$.

The former one verifies that the recipient's balance is payable (e.g., has a fallback function) and 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 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. 2.

Rules [METH*] of Fig. 2 deal with function invocations, which are particularly complex in mSCL. Rule [METH] defines successful non-payable function invocations $e.m(e')$. 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 callee'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 a mSCL program $\mathcal{P} = (C_1, \ldots, C_n)$ is a term

$$\prod_{i \in \mathcal{I}} C_i(\ell_i, \ell'_{C_i}) : \mathit{m}(C)$$

where $\bot$ is a dummy support contract name, $C, C' \in \mathit{names}(\mathcal{P})$ and $C_i$ is the contract name of $C_i$. That is what 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 $\mathit{Id}$ contains a dummy name $\mathit{User}$ 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 $\mathit{m}(\mathit{value}(\nu) \theta \mathcal{P})$ 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

$$\text{Bank}(\ell_{\ell} \ell_{\ell}; \ell_{\ell}) : \mathit{m}_{\text{pay}}.\mathit{value}(1)$$

where $\ell_{\ell} = \{\mathit{balance} = \nu\}$ and $\ell_{\ell} = \{\mathit{balance} = 1\}$.

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 a mSCL program and $S$ be an initial state such that $S \xrightarrow{\cdot} \mathcal{S}$. Then

1. Determinism: there is at most one $\mathcal{S}'$ such that $\mathcal{S} \xrightarrow{\cdot} \mathcal{S}'$.
2. Progress: either $\mathcal{S} \xrightarrow{\cdot} \mathcal{S}'$ 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 $\{e\}_{C,e,C,e}$ return a value and that when $e$ is of type address then $\{e\}_{C,e,C,e} = C'$ 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 backtrace 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 a 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. As we will see, the evaluation of a program amounts to compute a final environment from the initial ones, which are identical, by passing updated current environments from one function invocation to another.

4.1. Environments

$\Gamma$, called environment, is a map $(\mathit{Id} \rightarrow \mathit{Fld} \rightarrow \mathit{E}) \cup (\mathit{Var} \rightarrow \mathit{E})$; we always shorten $\Gamma(C)(e)$ to $\Gamma(C,e)$ and use $\Gamma(C, f \rightarrow e)$ to denote the update to $e$ of the field $C.f$. Notice that environments return abstract terms (which are expressions in mSCL) rather than (integer) values. We also use two update operations on environments: $\Gamma[C.f \leftarrow \Gamma(C.f) + e]$ and $\Gamma[C.f \leftarrow e]$.

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 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

$$\mathit{C.m}(\Gamma_0, \Gamma_1, \nu, \mathcal{T}, \mathcal{H}) = \sum_{\mathit{Id} \in \mathcal{I}} (\mathcal{H} = D) \Theta_0$$

(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 $\Gamma_0$ and $\Gamma_1$ that are pure and with disjoint codomains. $\Gamma_0$ is the environment that has to be returned in case of backtrace; $\Gamma_1$ is the environment that must be updated by the function body in case of successful termination;

---

To improve readability, we denote with $e$ the expressions that occur in mSCL programs and with $\nu$ the same expressions when used as abstract values in the intermediate language. Equivalently, $e$ and $\nu$ range over the productions of two grammars E and $\mathcal{P}$ that are defined identically.
2. the remaining parameters, namely \( v, \Pi \) 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 \( \text{Id} \) 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 \( \Theta \) is

\[
\Theta ::= \Gamma \mid e.m(\Gamma, \Gamma', e', e, H) \mid \sum_{\phi_i} \Theta_i
\]

where \( \phi_i \) are boolean expressions that also contain predicates such as \( m \in C \) or \( m.\text{payable} \in C \). According to the syntax, a function may either return an environment, or invoke another function, or have a nondeterministic behaviour \( \sum_{\phi_i} \Theta_i \) that is regulated by a finite set of predicates \( \phi_1, \ldots, \phi_n \). The term \( \sum_{\phi_i} \Theta_i \) is an abbreviation for \( (\phi_1)_{\Theta_1} + \cdots + (\phi_n)_{\Theta_n} \) (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 \( \Gamma' \) that is a ground environment with the formal one \( \Gamma \) that is a pure environment. We denote with \( \sigma_{\Gamma' \Gamma} \) the unique substitution such that \( \sigma_{\Gamma' \Gamma} \circ \Gamma = \Gamma' \).

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

\[
\begin{align*}
\text{Apply} & : \quad C.m(\Gamma_0, \Gamma_1, x, \Pi, H) = \sum_{1 \leq i \leq n} (H = D_i)\Theta_x \\
\quad & \in \Theta \quad \gamma = \psi \\
\quad & C.m(\Gamma, \Gamma', e, e', D_i) \Rightarrow \forall \Theta_{x_i} \{ (u, \sigma_{x_i} \Pi, \gamma) \circ \sigma_{\gamma' \gamma} \} \sum_{1 \leq i \leq n} \Theta_i
\end{align*}
\]

where \( \{ [e] \} \) is the value of \( e \). (The Definition of \( \{ [e] \} \) is omitted because straightforward.) We notice that the semantics of the intermediate programs is nondeterministic: if \( \Theta = (1 > 0)\Theta_1 + (2 > 1)\Theta_2 \) then it may evolve into either \( \Theta_1 \) or \( \Theta_2 \). 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

\[
\begin{align*}
\Gamma_0 & = [\text{Bank} \rightarrow \text{balance} \rightarrow x_{\text{bank}}] , \text{Thief} \rightarrow [\text{balance} \rightarrow x_{\text{thief}}] \\
\Gamma_1 & = [\text{Bank} \rightarrow \text{balance} \rightarrow y_{\text{bank}}] , \text{Thief} \rightarrow [\text{balance} \rightarrow y_{\text{thief}}]
\end{align*}
\]

Notice that \( \Gamma_0 \) and \( \Gamma_1 \) are pure environments with disjoint codomains. Let also \( \text{Id} = (\text{Bank}, \text{Thief}) \). For Bank.pay we obtain:

\[
\text{Bank.pay}(\Gamma_0, \Gamma_1, v, n, H) = \sum_{D_i} (H = D_i) \begin{cases} v \geq 1 \land y_{\text{bank}} > n \land n < 5 \Theta \\ + (v \geq 1 \land y_{\text{bank}} > n \land n < 5) \Gamma_1 \end{cases}
\]

where \( \Theta = (y_{\text{bank}} > n \land \text{fallback} \in D) \Theta + (y_{\text{bank}} \leq n) \Gamma_0 + (\text{fallback} \notin D) \Gamma_0 \Theta = (\text{ack} \in D) D.\text{ack}(\Gamma_0, \Gamma_1, \text{Bank}) + (\text{ack} \notin D \land \text{payable} \in D \land \text{fallback} \in D) \Gamma_1 + (\text{ack} \notin D \land \text{payable} \notin D \land \text{fallback} \notin D) \Gamma_0 \Gamma_1 = \Gamma_1 [\text{Bank.balance} \rightarrow n, D.\text{balance} \rightarrow n]
\]

For Thief.ack we get:

\[
\begin{align*}
\text{Thief.ack}(\Gamma_0, \Gamma_1, v, H) & = \sum_{D_i} (H = D) \Theta' \\
\text{where} \quad & \Theta' = (\text{pay} \in D) \Gamma_0 + (\text{pay} \in D \land y_{\text{thief}} \geq 1) \Theta' + (\text{pay} \notin D \land y_{\text{thief}} < 1) \Gamma_0 + (\text{pay} \notin D \land \text{payable} \notin D \land \text{fallback} \notin D \land y_{\text{thief}} \geq 1) \Gamma' + (\text{pay} \notin D \land \text{payable} \notin D \land \text{fallback} \notin D \land y_{\text{thief}} < 1) \Gamma_0 + (\text{pay} \notin D \land \text{payable} \notin D \land \text{fallback} \notin D) \Gamma_0
\end{align*}
\]

As regards the semantics of the intermediate language, let us discuss the transitions of Bank.pay(\( \Gamma, \Gamma' \), 1, 2, Thief), where \( \Gamma = [\text{Bank} \rightarrow \text{balance} \rightarrow 4], \text{Thief} \rightarrow \text{balance} \rightarrow 1] \).

\[
\begin{align*}
\text{Bank.pay}(\Gamma, \Gamma', 1, 2, \text{Thief}) & \Rightarrow_2 (1 \geq 1 \land 4 \geq 1 \land 2 < 5) \Theta' [\text{Thief}/D] [2.4.1/n_{\text{bank}}, y_{\text{bank}}] \\
& \Rightarrow_2 (\text{Thief} \rightarrow \text{Thief.ack}(\Gamma', 0, \text{Bank}) + (\text{ack.\text{Thief}} \rightarrow \Theta' [\text{Thief}/D] [2.4.1/n_{\text{bank}}, y_{\text{bank}}] \\
& \Rightarrow_2 (\text{Thief} \rightarrow \text{Thief.ack}(\Gamma, 0, \text{Bank}) + (\text{ack.\text{Thief}} \rightarrow \Theta' [\text{Thief}/D] [2.4.1/n_{\text{bank}}, y_{\text{bank}}] \\
& \Rightarrow_2 (\text{Thief} \rightarrow \Theta' [\text{Bank}/D] [2.4.1/n_{\text{bank}}, y_{\text{bank}}]) \\
& \Rightarrow_2 \text{Bank.pay}(\Gamma, \Gamma', 1, 2, \text{Thief})
\end{align*}
\]

where

\[
\begin{align*}
\Gamma' & = [\text{Bank} \rightarrow [\text{balance} \rightarrow 2], \text{Thief} \rightarrow [\text{balance} \rightarrow 3]) \\
\Gamma'' & = [\text{Bank} \rightarrow [\text{balance} \rightarrow 3], \text{Thief} \rightarrow [\text{balance} \rightarrow 2])
\end{align*}
\]

### 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**: \( \Gamma, \phi_1, \phi_2 : e', \phi_3, \phi_4 : e' \), where \( e, e' \) 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: \( \Gamma \vdash ^c e, S : \Theta \), where \( \Gamma \) is the backtracking environment, \( \Gamma' \) is the current environment and \( \Theta \) is the resulting intermediate code (\( e, C \) and \( D \) are similar to the corresponding one for judgments of expressions). Backtracking and current environments correspond to the (instances of) environments \( \Gamma_0 \) and \( \Gamma_1 \) in the function definitions and are used to model backtracking (in case of failures) and success, respectively.

The translation of expressions is reported in Fig. 3. It partially evaluates expressions by replacing access 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 \( \Gamma \) (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 \( H \in Id \).

Rule [FUNCTION] 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 \textit{Bank-pay} and \textit{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 a \( mSCL \) program and its intermediate code. The following Definition intends to specify this relationship.

Definition 4. (Correspondence of states and intermediate codes)

Given a state \( S = \prod_{i \in 1..n} C_i(\ell_i \cdot e_i) | D \cdot S_0 : S \), we define

\[
envs(S) \overset{\text{def}}{=} \{ [C_i, \ell_i \cdot e_i] \mid i \in 1..n \} \cup \{ [C_n, \ell_n \cdot e_n] \mid \ell_n = \ell_{S_0} | \ell_{S_0} \}
\]

and we write \( S \vdash \Theta \) whenever \( S = \prod_{i \in 1..n} C_i(\ell_i \cdot e_i) | D \cdot S_0 : S \) and \( envs(S) = \Theta \cdot S \) or \( S = \prod_{i \in 1..n} C_i(\ell_i \cdot e_i) | D \cdot S_0 : \Theta \) and \( \Theta = \Gamma \) and \( envs(S) = \Gamma \).

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

Theorem 5. Let \( P \) be a \( mSCL \) program such that \( \vdash P : I \) and let \( S \) be an initial state such that \( \vdash S \Theta \). Then

1. (determinism) If \( \Theta \Rightarrow \Theta' \) then there is at most one \( \Theta'' \) such that \( \Theta \Rightarrow \Theta'' \);
2. (correctness) If \( \Theta \Rightarrow \Theta' \) then there is a \( \Theta' \) such that \( \Theta \Rightarrow \Theta' \).

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 to notice 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 into symbolic upper bounds expressed as functions on the value of the fields of the initial environment and the actual parameters of the initial call.

![Fig. 3. Translation of mSCL expressions.](image-url)
We start by defining the final gain and loss associated to a smart contract $C_0$, a system of equations $I$ and an initial invocation $C_m$:

$$\Gamma \vdash \text{gain}(\Gamma, \Gamma', \Gamma, [D, x \mapsto e'] ; C_0) \vdash \Gamma'$$

$$\Gamma \vdash \text{loss}(\Gamma, \Gamma', \Gamma, [D, x \mapsto e'] ; C_0) \vdash \Gamma'$$

$\Gamma_0 : \text{mSCL} \rightarrow \text{mSCL}$

$\text{m-value}(E, E') : (\max(0, \Gamma_0(\text{balance}[x])) : \Gamma')$

$\text{m-value}(E, E') : (\max(0, \Gamma_0(\text{balance}[x])) : \Gamma')$

$$\text{transfer}(E, E') \vdash \text{m-value}(E', E)$$

$$\text{m-value}(E, E') : (\max(0, \Gamma_0(\text{balance}[x])) : \Gamma')$$

$$\text{m-value}(E, E') : (\max(0, \Gamma_0(\text{balance}[x])) : \Gamma')$$

Fig. 4. The translation of mSCL statements.

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

We start by defining the final gain and loss associated to a smart contract $C$, a system of equations $I$ and an initial invocation $C_m(\Gamma, \Gamma, x, X, H)$.

6. Let $P$ be a mSCL program and $\vdash P : I$. Let $\text{gain}_{C_m}(\Gamma, x, I, H)$ and $\text{loss}_{C_m}(\Gamma, x, I, H)$ be the functions

$$\text{gain}_{C_m}(\Gamma, x, I, H) = \begin{cases} \max(0, \Gamma'(\text{balance}) - \Gamma(\text{balance})) & \text{if } C_m(\Gamma, x, I, H) \vdash \Gamma' \\ 0 & \text{otherwise} \end{cases}$$

$$\text{loss}_{C_m}(\Gamma, x, I, H) = \begin{cases} \max(0, \Gamma(\text{balance}) - \Gamma'(\text{balance})) & \text{if } C_m(\Gamma, x, I, H) \vdash \Gamma' \\ 0 & \text{otherwise} \end{cases}$$

where, for every $D \in \text{enames}(P)$, $\text{dom}(\Gamma(D)) = \text{fields}(D)$. 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 $\text{gain}_{C_m}$ and $\text{loss}_{C_m}$ compute the amount of cryptocurrency gained/lost at the end of the computation of $C_m(\Gamma, x, I, H)$. 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 $\text{loss}_{C_m}$ is not the opposite of $\text{gain}_{C_m}$. 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 $\text{gain}_{C_m}$ and $\text{loss}_{C_m}$. However,
sometimes our technique returns asymptotic upper bounds that are less informative, like in Example.

**Definition 7.** A function \( \text{ugain}^C_{\text{cl}, m} \) is an upper bound of \( \text{GAIN}^C_{\text{cl}, m} \) if and only if, for every \( \Gamma, \nu, \varphi, D \) in the domain of Definition of \( \text{GAIN}^C_{\text{cl}, m} \), \( \text{ugain}^C_{\text{cl}, m}(\Gamma, \nu, \varphi, D) \leq \text{GAIN}^C_{\text{cl}, m}(\Gamma, \nu, \varphi, D) \).

- A function \( \text{ugain}^C_{\text{cl}, m} \) is an asymptotic upper bound of \( \text{GAIN}^C_{\text{cl}, m} \) if and only if \( \text{GAIN}^C_{\text{cl}, m} \in \mathcal{O}(\text{ugain}^C_{\text{cl}, m}) \).
- Similarly for \( \text{LOSS}^C_{\text{cl}, m} \).

Definitions 6 and 7 are given on 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 a mSCL program. Let \( \text{mGAIN}^C_D \) and \( \text{mLOSS}^C_D \) be the functions defined on initial states \( S \):

\[
\text{mGAIN}^C_D(S) = \begin{cases} \max(0, C.\text{balance}(S)) - C.\text{balance}(S) & \text{if } S \rightarrow^* S' \text{ for some } S' \text{ final} \\ 0 & \text{otherwise} \end{cases}
\]

\[
\text{mLOSS}^C_D(S) = \begin{cases} \max(0, C.\text{balance}(S)) - C.\text{balance}(S) & \text{if } S \rightarrow^* S' \text{ for some } S' \text{ final} \\ 0 & \text{otherwise} \end{cases}
\]

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

**Corollary 1.** (of Theorem 5) Let \( S = \prod_{i=1}^n C_i(\ell_i, \varepsilon_i) \mid \mathbb{D} : C.m(\mathbb{P}) \) be an initial state of a mSCL program \( P \) and \( \rightarrow^* : I \) and \( S \vdash C.m(\Gamma, 0, \mathbb{X}, D) \), where \( (\Gamma, \Gamma) = \text{env}(S) \).

\[ \text{Then } \quad \text{mGAIN}^C_D(S) = \text{GAIN}^C_{\text{cl}, m}(\Gamma, 0, \mathbb{X}, D) \quad \text{and} \quad \text{mLOSS}^C_D(S) = \text{LOSS}^C_{\text{cl}, m}(\Gamma, 0, \mathbb{X}, D). \]

As a consequence of Corollary 1, instead of computing upper bounds of \( \text{mGAIN}^C_D(S) \) and \( \text{mLOSS}^C_D(S) \), 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 [14] and PUBS [1]. This will allow us to compute \( \text{mGAIN}^C_D \) and \( \text{mLOSS}^C_D \) automatically, without any difficulty.

In the following we introduce the syntax of CoFloCo and we define the set of CoFloCo cost equations associated to a program in our intermediate language such that the cost model considered by CoFloCo is either that of \( \text{GAIN}^C_{\text{cl}, m} \) or that of \( \text{LOSS}^C_{\text{cl}, m} \).

5.2. The syntax and semantics of CoFloCo

Cost equation solvers take a list of equations in input that are terms [14].
\[ m(\tau) = e + \sum_{m_i(\tau)} \phi \]

where variables occurring in the right-hand side and in \( \phi \) are a subset of \( \Sigma \) and \( \Sigma' \):

- \( m \) is a (cost) function symbol,
- \( e \) (i.e. the cost of the step) and \( \Sigma \) are Presburger arithmetic expressions, namely (\( q \) is a positive rational number)
  \[ e ::= x \mid q \mid e + e \mid e - e \mid q^* e \mid \max(e_1, \ldots, e_k) \]

- \( \phi \) is a conjunction of linear constraints, e.g. constraints of the form \( \ell_1 < \ell_2 \) or \( \ell_1 \leq \ell_2 \) or \( \ell_1 = \ell_2 \), where both \( \ell_1 \) and \( \ell_2 \) are Presburger arithmetic expressions.

The solution of a cost program is the computation of bounds of a particular function symbol (typically the one of the first equation in the list). The bounds are parametric in the formal parameters of the function symbol. The operational semantics of the (subset of) \( \Sigma \) list). The bounds are parametric in the formal parameters of the function symbol.

\[ \text{The operational semantics in functional style.} \]

\[ \text{The bounds are parametric in the formal parameters of the function symbol.} \]

\[ \phi \]

\[ \text{The operational semantics of the (subset of) \( \Sigma \) CoFloCo we are considering is defined below.} \]

\[ \text{Definition 9. (Semantics of cost equations seen as a functional language) Let} \]

\[ \rightarrow_{\text{CoFloCo}} \text{be the reduction relation over ground Presburger expressions augmented with function calls (in the obvious way) defined by the following two rewriting rules, that can be applied in any context:} \]

1. \[ m(\Sigma) \rightarrow_{\text{CoFloCo}} \phi(\Sigma) + \sum_{e \neq e'} m_i(\Sigma) \text{ for every cost equation} \]
2. \[ e \rightarrow_{\text{CoFloCo}} \phi \text{ if } e \text{ is a Presburger expression whose value is } \phi. \]

\[ \rightarrow_{\text{CoFloCo}} \text{ as a reduction relation, is obviously non-deterministic, as the following example shows. However, all cost equations generated from } \text{mSCL programs exhibit a deterministic behaviour.} \]

\[ \text{Example 10. Consider the following set of cost equations:} \]

\[ n(x) = x + 1 \]
\[ m(x) = 1 + m(2x) \quad [0 \leq x] \]
\[ m(x) = 2 - m(2x) \quad [0 \leq x] \]

It turns out that \( m(1) \rightarrow_{\text{CoFloCo}} 4 \) and \( m(1) \rightarrow_{\text{CoFloCo}} -1 \).

\[ \text{5.3. The translation} \]

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 \( \Gamma \) amounts to compute the empty set of calls where the step has cost max(0, \( \Gamma(C.\ balance) - \Gamma(C.\ balance) \)) — to compute GAIN\( _{\Gamma} \) — or max(0, \( \Gamma(C.\ balance) - \Gamma(C.\ balance) \)) — to compute LOSS\( _{\Gamma} \).

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

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

- the simplification of a formula \( \phi \), written \( \Rightarrow \phi \), is an homomorphic operator with respect to all arithmetic operators but subtraction and such that

\[ \Rightarrow e - \Rightarrow e' = \max(\Rightarrow e - \Rightarrow e', 0) \]
\[ \Rightarrow e_1 \oplus e_2 = \max(e_1, e_2) \]
\[ \Rightarrow \text{if } x \notin D \]
\[ \Rightarrow \text{if } D \]
\[ \Rightarrow m \in D \Rightarrow \min(\Rightarrow m, \Rightarrow D) \]
\[ \Rightarrow m.\ payable \in D \Rightarrow \min(\Rightarrow m.\ payable, \Rightarrow D) \]
\[ \Rightarrow m \notin D \Rightarrow \max(\Rightarrow m, \Rightarrow \neg D) \]
\[ \Rightarrow m.\ payable \notin D \Rightarrow \max(\Rightarrow m.\ payable, \Rightarrow \neg D) \]
\[ \Rightarrow \text{fallback } \neq D \Rightarrow \text{true if } D \text{ declares the fallback function, } \text{false otherwise} \]

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

Moreover, \( \Rightarrow \) also puts formulae \( \phi \) in disjunctive normal form plus the additional constraint that atomic formulae are inequalities. For example, \( e \neq e' \) is normalized to \( e < e' \lor e' < e \).

- the flattening operation on environments \( \Gamma \), noted \( [\Gamma] \), encodes \( \Gamma \) into a list of integer expressions.

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

\[ \text{In the rest of the section we use the green color both for cost equations and for simplified intermediate programs, whose syntax is looser since it does not require formulae to be only conjunctions.} \]
let $\Gamma = \left[ \begin{array}{l}
C_1 \mapsto [e_{1,1} \mapsto e_{1,1}, \ldots, e_{1,n} \mapsto e_{1,n}, \text{balance} \mapsto e_{1,b}], \\
\ldots, C_l \mapsto [e_{l,1} \mapsto e_{l,1}, \ldots, e_{l,n} \mapsto e_{l,n}, \text{balance} \mapsto e_{l,b}] 
\end{array} \right]
$

according to total orders $C_i \leq C_{i+1}$ and $f_{ij} \leq f_{i,j+1}$, then

$[\Gamma] \triangleq \langle \langle e_{1,1}, \ldots, e_{1,n}, e_{1,1}, \ldots, e_{1,n}, \ldots, e_{n,n}, e_{n,n} \rangle \rangle$

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 $\gamma \rightarrow \gamma'$ is lifted to the intermediate code as follows:

$\gamma \rightarrow \gamma' = [\Gamma]$

where $\gamma' = e.m(\Gamma_1, \Gamma_2, e, \ldots, D^\gamma) = e.m(\Gamma_1, \Gamma_2, e, \ldots, e^\gamma, \ldots, D^\gamma)$

$\sum_{\gamma \in \Gamma} (\gamma e^\gamma \Gamma e^\gamma) \sum_{\gamma \in \Gamma} (\gamma e^\gamma \Gamma e^\gamma)
$

- The simplification $\gamma \rightarrow \gamma'$ of a program, i.e. a list of function definitions, is obtained by simplifying each function in the list as follows

$\gamma \rightarrow \gamma' = \sum_{\gamma \in \Gamma} (\gamma e^\gamma \Gamma e^\gamma)
$

A simplified intermediate program $\ell$ is turned into the sets of cost equation ($\ell$) as follows: every simplified function declaration

$C.m(\Gamma_0, \Gamma_1, v, \tau, H) = \sum_{\gamma \in \Gamma} \left( \bigvee_{\gamma \in \Gamma} \phi \right) \Theta_{\gamma}
$

where

- $\phi$ is a disjunction of comparisons between Presburger expressions

$C.m(\Gamma_0, \Gamma_1, v, \tau, H) = \text{cost}(\Gamma_0, \Theta_0) \left[ \phi \right]
$

$C.m(\Gamma_0, \Gamma_1, v, \tau, H) = \text{cost}(\Gamma_0, \Theta_0) \left[ \phi \right]
$

$C.m(\Gamma_0, \Gamma_1, v, \tau, H) = \text{cost}(\Gamma_0, \Theta_0) \left[ \phi \right]
$

$C.m(\Gamma_0, \Gamma_1, v, \tau, H) = \text{cost}(\Gamma_0, \Theta_0) \left[ \phi \right]
$

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

$\text{main}(\Gamma, v) = C.m(\Gamma, \Gamma, v)
$

$b_1, \ldots, b_n$ are the variables in $[\Gamma], \text{v}$ of type unit. We assume that these variables are non-negative (this is required because variables in CoFloCo are signed).

To conclude, if $T$ is a program in the intermediate language and $C.m(\Gamma, \tau, H)$ its initial state, then the Eq. (1) plus ($\Gamma^T$) gives the bunch of CoFloCo cost equations.

**Example 11.** To illustrate the output of our technique we compute the cost equations of the functions $\text{Bank.pay}$ and $\text{Thief.ack}$ in Example 3, according to the cost model that computes the loss of the Bank. We shorten $\text{Bank}$ and $\text{Thief}$ into $B$ and $T$, respectively, for readability sake, we always write predicates such as $\text{fallback} \in T$ and $\text{ack} \in T$, 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 is always false (e.g. $\text{pay} \in T$) are not shown nor generated by our translator.

\[
\begin{align*}
\text{main}(x_{gb}, x_{gb}, y_{gb}, v, n, H) & = B.\text{pay}(x_{gb}, x_{gb}, x_{gb}, y_{gb}, v, n, H) \\
& = \begin{cases} 
0 & x_{gb} \geq 0 \land x_{gb} \geq 0 \land v \geq 0 \land n \geq 0 \\
\text{fallback} \in T \land \text{ack} \in T & H = T \land v \geq 1 \land y_{gb} > n \land n < 5 \land \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
B.\text{pay}(x_{gb}, y_{gb}, y_{gb}, y_{gb}, y_{gb}, v, n, H) & = \text{max}(0, x_{gb} - y_{gb}) \\
& = \begin{cases} 
\text{fallback} \in B \land \text{ack} \in B \land \text{pay.ack} \in B & H = T \land v \geq 1 \land y_{gb} \geq 0 \land n < 5 \land \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
B.\text{pay}(x_{gb}, y_{gb}, y_{gb}, y_{gb}, y_{gb}, y_{gb}, v, n, H) & = \text{max}(0, x_{gb} - y_{gb}) \\
& = \begin{cases} 
\text{fallback} \in B \land \text{ack} \in B \land \text{pay.ack} \in B & H = T \land v \geq 1 \land y_{gb} \geq 0 \land n < 5 \land \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{main}(x_{gb}, x_{gb}, y_{gb}, v, n, H) & = B.\text{pay}(x_{gb}, x_{gb}, x_{gb}, y_{gb}, v, n, H) \\
& = \begin{cases} 
0 & x_{gb} \geq 0 \land x_{gb} \geq 0 \land v \geq 0 \land n \geq 0 \\
\text{fallback} \in T \land \text{ack} \in T & H = T \land v \geq 1 \land y_{gb} > n \land n < 5 \land \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
T.\text{ack}(x_{gb}, y_{gb}, y_{gb}, y_{gb}, y_{gb}, v, n, H) & = \text{max}(0, x_{gb} - y_{gb}) \\
& = \begin{cases} 
\text{fallback} \in B \land \text{ack} \in B \land \text{pay.ack} \in B & H = T \land v \geq 1 \land y_{gb} \geq 0 \land n < 5 \land \\
\end{cases}
\end{align*}
\]

\[
\begin{align*}
T.\text{ack}(x_{gb}, y_{gb}, y_{gb}, y_{gb}, y_{gb}, v, n, H) & = \text{max}(0, x_{gb} - y_{gb}) \\
& = \begin{cases} 
\text{fallback} \in B \land \text{ack} \in B \land \text{pay.ack} \in B & H = T \land v \geq 1 \land y_{gb} \geq 0 \land n < 5 \land \\
\end{cases}
\end{align*}
\]

In Section 6 we analyze CoFloCo [14] 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 $P$ be a scs program such that $\vdash P : T$ and let $S$ be an initial state and $S \vdash C.m(\Gamma, v, \tau, H)$ and $C.m(\Gamma, v, \tau, H) \Rightarrow \Gamma^T$. Let us extend ($\Gamma^T$) (where we use either $\text{cost.\text{gain}}$ or $\text{cost.\text{los}}$ during the translation) with a main function that calls $C.m$. Then

1. **Determinism:** $\text{main}(\Gamma, v, \tau, H)$ has a unique $\rightarrow_C$-normal-form

2. **Correctness:**

- $\text{main}(\Gamma, v, \tau, H) \rightarrow_C \text{CoFloCo} \text{GAIN}_C(\gamma, v, \tau, H)$ if we selected $\text{cost.\text{gain}}$ during the translation,

- $\text{main}(\Gamma, v, \tau, H) \rightarrow_C \text{CoFloCo} \text{LOST}_C(\gamma, v, \tau, H)$ if we selected $\text{cost.\text{los}}$ 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 [14].

**Conjecture 1.** (Correctness of CoFloCo) Given a set of guarded cost equations whose first equation is
main(φ) = m(φ) \quad |φ|

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

**Theorem 13. (Final theorem)** Let \( S \) be an initial state of a \( \text{mSCL} \) program \( S \vdash P : I \) and \( S \vdash C.m(\Gamma, \nu, v, \gamma, H) \). If CoFloCo claims \( f \) to be an upper bound/an asymptotic upper bound to main of the equations obtained by ("\( \Gamma \)"), with a main function that calls \( C.m \), then \( f \) is an upper bound/an asymptotic upper bound to \( mGAIN(S) \), if \( cost_{\text{Gain}}^{f} \) was selected during the translation, or to \( mLOSS(S) \) if \( cost_{\text{Loss}}^{f} \) was.

**Proof.** Either \( S \) converges or it diverges. If it diverges, then both \( mGAIN(S) \) and \( mLOSS(S) \) are defined to be 0, and the statement trivially holds because all bounds computed by CoFloCo for our cost equations are non negative.

If \( S \) converges then, by Theorem 2, the last reduction step leads to a final state \( S' \). Thus, by Definition 4 and Theorem 5, \( C.m(\Gamma, \nu, v, \gamma, H) \Rightarrow \Gamma' \) where \( \text{ens}(S') = \Gamma', \Gamma' \).

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

Note that CoFloCo may also compute a finite upper bound also for diverging \( \text{mSCL} \) 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 cannot diverge. However, in our case, the cost equations compute 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 \( \text{mSCL} \) in about 2500 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/behaveial_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 to notice 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 \( \text{mSCL} \). In Table 1, we report our analysis of four archetypal contracts we identified among the other ones. In particular, for every program, we give the lines of original code (\( \# \text{LOC} \)), those produced by our translation (\( \# \text{LMC} \)), the number of equations produced (\( \# \text{Equations} \)) 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 (\( \text{Bank__balance} \)), Thief’s balance (\( \text{Thief__balance} \)), the invokes of the analyzed function (\( \text{msg.sender} \)), the amount of coins passed to the function (\( \text{msg.value} \)) and the function parameter (N). CoFloCo’s output is:

```plaintext
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 \( \text{nat}(x) \) 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 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 currently winning one, the previous winner is refunded. Otherwise the bid is refunded to the sender. The result of the analysis is interesting:
The bidder cannot gain any money by bidding: either he can lose all its bid \( \text{nat}(\text{Bid1}) \) (because he is winning the auction) or he can lose the lesser amount \( \text{nat}(-\text{Auctioneer\_max} + \text{Bid1}) \) because he was already winning and decided to lift his offer (the previous offer is returned back).

In the “Handover Ponzi scheme” of [5], 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 the 90% is used to pay the previous user.

We analyze two scenarios. The first scenario is when Player joins the scheme, followed by Player 2. The analysis yields:

**MAXIMUM GAIN:**
Maximum cost of \( \text{main\_}(\text{Player\_balance\_}, \text{Player\_amount}, \text{Player2\_balance\_}, \text{N}, ...) : \text{nat}(7/20\times N) \)
Asymptotic class:\( n \)

**MAXIMUM LOSS:**
Maximum cost of \( \text{main\_}(\text{Player\_balance\_}, \text{Player\_amount}, \text{Player2\_balance\_}, \text{N}, ...) : \text{0} \)
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 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.

**MAXIMUM GAIN:**
Maximum cost of \( \text{main\_}(\text{Player\_balance\_}, \text{Player\_amount}, \text{N}, ...) : \text{0} \)
Asymptotic class: constant

**MAXIMUM LOSS:**
Maximum cost of \( \text{main\_}(\text{Player\_balance\_}, \text{Player\_amount}, \text{N}, ...) : \text{nat}(N) \)
Asymptotic class: \( n \)

Table 1

<table>
<thead>
<tr>
<th></th>
<th>( # \text{LOC} )</th>
<th>( # \text{LMC} )</th>
<th>( # \text{Equations} )</th>
<th>CoFloCo’s Time for gain + loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank-Thief code</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>734 ms + 240 ms</td>
</tr>
<tr>
<td>English Auction Scheme</td>
<td>32</td>
<td>32</td>
<td>38</td>
<td>509 ms + 468 ms</td>
</tr>
<tr>
<td>Handover Ponzi Scheme</td>
<td>42</td>
<td>50</td>
<td>336</td>
<td>6,964 ms + 4,784 ms</td>
</tr>
<tr>
<td>Chain-shaped Ponzi scheme</td>
<td>45</td>
<td>63</td>
<td>1030</td>
<td>27,978 ms + 27,962 ms</td>
</tr>
</tbody>
</table>

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

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 in 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 addresses becomes.
(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 solution remains the same: the translation of each function body must deal with local variables and function bodies in order to cope with every possible instance of the variable.

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 parameterising our translation with a constant $\kappa$ that limits the length of the stack. The value $\kappa$ 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 $\kappa$ as the maximal weight of a path;
   (b) otherwise the value for $\kappa$ is requested to the user by the analyzer. In this case we are technically verifying the analyzer, that reverts if it tries to nest more than $\kappa$ non tail-recursive calls, exhausting the stack space.

Let $i$ be the maximum number of parameters and local variables of functions. The maximal size of the stack is bounded by $(i + 4) \times (\kappa + 1)$ where $i + 4$ 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 $\kappa$ is required for the additional stack frame used for the initial call to the program.

More formally, let $\sigma$ be a sequence of frames of the form

$$a_1 \ldots a_k \otimes \ldots \otimes$$

with $\kappa + 1$ frames.

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.
where

- the initial frames \( a_i \) are equal to \( \langle C, m, D, e \rangle \), where \( C \) is the callee, \( m \) is the callee’s function to be executed with arguments \( \tau \), assuming it has been called by \( D \) that has transferred \( e \) cryptocurrencies;
- the frames \( \perp_i \), called empty frames, are equal to \( \langle \perp, m, \perp, \perp, \perp \rangle \), where \( \perp, m_\perp \) 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 \( \sigma \) is always empty;
- a frame \( \langle C, m, \tau, D, e \rangle \) is equal to \( \perp \) if and only if \( C = \perp \).

A sequence \( \sigma \) 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:

- \( |e| \) returns the length of the tuple;
- \( \tau \rightarrow D \) returns the prefix of \( \tau \) whose length is equal to the number of arguments of \( D; m \);
- \( \tau \rightarrow f \) drops the last frame in \( \sigma \), i.e. \( \sigma \rightarrow f = \sigma \);
- \( [\sigma] \rightarrow f \) returns the first element of the last-but-one frame in \( \sigma \), if it exists, or any value different from \( \perp \), if \( k = 0 \). That is \( [\sigma] \rightarrow f = C \). The key property is that if \( [\sigma] \rightarrow f = \perp \) then the last-but-one frame is unused (it is \( \perp \)), therefore the sequence \( \sigma \) is not full and it can hold one more element. It is sufficient to throw away the last empty frame (using \( \cdot \rightarrow f \)) and push the new element at the beginning, effectively shifting to the right all the already present frames.

Finally, we write \( m \in \text{fun}(C) \) to mean that \( C \in \text{names}(\mathcal{P}) \) 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 $\sigma$ of the form $\ldots \sigma \ldots$ – the stack is empty – or not. According to our modelling, the former case is when $H = \bot$, where $H$ is the first element of the initial frame. In this case we return 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 $e_i(\ldots)$ that sets $msg.sender$ to $D'$ and $msg.value$ to $e_i$. However, this also transfers again $e_i$ units of cryprocurrency from $msg.sender$ to the receiver (see rule [NVK-TAIL]), which would not be correct. To contrast this, we perform the translation in $\Gamma_1^i$ where we first transfer back $e_i$ units from the receiver to $msg.sender$.

\[
\Gamma_0 = [\text{Bank} \rightarrow \text{balance} \rightarrow x_{\text{Bank} \cdot \text{to}} \cdot \text{to_pay} \rightarrow x_{\text{Bank} \cdot \text{to PAY}}] \\
\Gamma_1 = [\text{Bank} \rightarrow \text{balance} \rightarrow y_{\text{Bank} \cdot \text{to}} \cdot \text{to_pay} \rightarrow y_{\text{Bank} \cdot \text{to PAY}}], \text{Thief} \rightarrow \text{balance} \rightarrow x_{\text{Thief} \cdot \text{to}}]
\]

We notice that in the premise of [EMPTY-CONT] we pass two expressions of the intermediate language ($e_i'$ and $e_i''$) where expressions of mSCL are expected. This is correct since, according to the rules in Fig. 3, $e_i'$ and $e_i''$ 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 on the sequence of frames takes the first argument that is provided by the return and is therefore taken by $mS$. Rules [ASSIGN-CONT-NVK] 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 environment (e.g. dom($\Gamma_1^i$[VAR]), see also rule [FUNCTION-CONT])) plus an additional variable for function 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 [NVK-TAIL-NV] and [NVK-TAIL] extend rules [NVK-NV] and [NVK] 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 [25]:

```solidity
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 circumvent 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 $\text{Bank.pay}(1)$ by Thief is to drain the account.

Note that the graph of invocations is cyclic; therefore our technique analyzes the $n$-th approximant and compute 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 of the results. The following intermediate code defines the approximant 1 of the DAO-like attack.

Let $\Gamma_0$ and $\Gamma_1$ two pure environments (with disjoint codomains) defined as follows:

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

- $\sigma = s_1, ..., s_{12}$
- 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 automated using standard techniques (e.g. abstract interpretation);
- $H/f$ is always the value of $msg.sender$ and $v$ the value of $msg.value$;
- in Bank.pay:

$$
\begin{align*}
\Gamma_1^i & = \Gamma_0^{[\text{Bank.balance} \rightarrow -n, \text{Thief.balance} \rightarrow +n]} \\
\sigma^i & = \text{pay_cont}, v, \text{Bank}, n, 0, s_1, s_2, s_3, s_4, s_5, s_6
\end{align*}
$$

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

```solidity
Bank.pay(\Gamma_0, \Gamma_1, H, v, n, \sigma) =
(H=Bank) ... -- deadcode: the Bank never calls Bank pay + (H=Thief)
(\Gamma_0^{[\text{Bank.pay_cont}]}, n, \sigma) --> if (this.balance > n & & n <= to_pay)
(s_1) -- stack not full
(n > 0) --> to_pay = to_pay - n;
-- msg.sender.transfer(\text{calls Thief.fallback Execution DISPATCH})
Thief.fallback(\Gamma_0, \Gamma_1, \text{Bank.n, Bank.'}s) + (n < 0) --> revert
(\Gamma_1) -- revert: push on full stack
\}
```

The function pay_cont implementing the continuation $\text{to_pay} = \text{to_pay} - n$ is:

```solidity
pay_cont(\Gamma_0, \Gamma_1, H, v, n, \sigma) = \text{runtime DISPATCH}
(\Gamma_0, \Gamma_1^{[\text{Bank.to_pay} \rightarrow -n]}, 0, 0, 0, 0, \sigma)
```

where $\text{runtime DISPATCH}$ 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
\[ \sigma' = s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12}, \ldots \]

\[
\text{runtime
dispatch}(\Gamma_0, \Gamma_1, H, v, r, \sigma) =
\]
\[
(s_1 = \text{Bank})
\]
\[
s_2 = \text{pay\_cont} \quad \text{pay\_cont}(\Gamma_0, \Gamma_1, s_1, s_2, s_3, s_4, s_5, s_6, \sigma')
\]
\[
+ (s_1 = \text{Thief}) (s_2 = \text{fallback}) \quad (H = \text{Thief}) \quad (\text{Thief\_fallback is never used as a continuation})
\]
\[
+ (s_1 = \bot) \quad \text{empty continuation stack: commit}
\]

Finally, the intermediate code for Thief\_fallback is:

\[
\text{Thief\_fallback}(\Gamma_0, \Gamma_1, H, v, \sigma) =
\]
\[
(H = \text{Bank}) \quad \text{Bank\_pay}(\Gamma_0, \Gamma_1, \text{Thief\_0.1}, \sigma) \quad \text{tail call to Bank\_pay}
\]
\[
+ (H = \text{Thief}) \quad \text{Thief never calls its fallback}
\]

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

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

1. for every function \( C.m(\Gamma_0, \Gamma_1, v, x, H, \sigma) = \Theta_{C.m}, \) let \( \bigwedge_{i \in 1..h(\phi)} \Theta_i \) be the canonical form of \( ^c\Theta_{C.m} \) (therefore every \( \psi \) is a conjunction). Then we have the following cost equations:

\[
C.m(\Gamma_0, \Gamma_1, v, x, H, \sigma) = \Theta_1 [\varphi_1]
\]

\[
C.m(\Gamma_0, \Gamma_1, v, x, H, \sigma) = \Theta_h [\varphi_h]
\]

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:

\[
main(\Gamma, \bar{y}) = C.m(\Gamma, \bar{y}, \bar{\Pi}, \ldots, \bar{\Pi}) \quad [b_1 \geq 0 \land \ldots \land b_n \geq 0]
\]

7.4. The analysis

We use the same arguments of 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 \((C_1, \ldots, C_n)\) be a program in the mSCL extended syntax, where \( \kappa \) is the maximal weight of a path in the graph of invocations and \( i \) is the maximal number of arguments and local variables of a function. For every sequence of \( \kappa + 1 \) frames \( \sigma = \alpha_1, \alpha_2, \ldots, \alpha_h, \bar{\Pi}, \ldots, \bar{\Pi} \), let \( \theta \) be the tuple whose length is \((i + 4) \times (\kappa + 1)\) that is defined as follows:

\[
\alpha_1, \ldots, \alpha_h, \bar{\Pi}, \ldots, \bar{\Pi} = \alpha_1, \ldots, \alpha_h, \bar{\Pi}, \ldots, \bar{\Pi}
\]

\[
\langle C.m, \bar{\varepsilon}, D, e' \rangle = C.m, \bar{\varepsilon}, \perp, \ldots, \perp, D, e'
\]

\[
\bar{\Pi} = \perp, \ldots, \perp
\]

\[
\text{\( i - \lvert \bar{\Pi} \rvert \) times}
\]

\[
\text{\( i + 4 \) times}
\]

Sequences of \((i + 4) \times (\kappa + 1)\) elements will be ranged over by \( \sigma \). Cost
\[
\text{main}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, H, v, n) = \text{bank\_pay}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, H, v, n, 12) = \begin{cases} 5 & x_{\text{Bank}, b} \geq 0 \land x_{\text{Thief}, b} \geq 0 \land v \geq 0 \land x_{\text{Bank, to\_pay}} = 0 \\
\end{cases}
\]

\[
\text{bank\_pay\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, n, \sigma) = \text{thief\_fallback}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, n, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b + n, 0, s_{1}, s_{2}, s_{3}, s_{4}, s_{5}, s_{6})
\]

\[
\text{Bank, n, Bank, pay\_cont\_int, int, v, Bank, n, 0, s_{1}, s_{2}, s_{3}, s_{4}, s_{5}, s_{6}}
\]

\[
[H = \text{Thief} \land y_{\text{Bank}, b} > n \land y_{\text{Bank, to\_pay}} \geq n \land s_{1} = 1 \land n \geq 0 \land y_{\text{Bank}, b} > n]
\]

\[
\text{bank\_pay\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, n, \sigma) = 0
\]

\[
[H = \text{Thief} \land y_{\text{Bank}, b} > n \land y_{\text{Bank, to\_pay}} \geq n \land s_{1} = 1 \land n < 0]
\]

\[
\text{bank\_pay\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, n, \sigma) = 0
\]

\[
[H = \text{Thief} \land y_{\text{Bank}, b} > n \land y_{\text{Bank, to\_pay}} \geq n \land s_{1} = 1 \land y_{\text{Bank}, b} < n]
\]

\[
\text{bank\_pay\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, n, \sigma) = 0
\]

\[
[H = \text{Thief} \land y_{\text{Bank}, b} > n \land y_{\text{Bank, to\_pay}} \geq n \land s_{1} < 1]
\]

\[
\text{bank\_pay\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, n, \sigma) = 0
\]

\[
[H = \text{Thief} \land y_{\text{Bank}, b} > n \land y_{\text{Bank, to\_pay}} \geq n \land s_{1} > 1]
\]

\[
\text{bank\_pay\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, n, \sigma) = \text{runtime\_dispatch\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, 0, 0, 0, \sigma)
\]

\[
[H = \text{Thief} \land y_{\text{Bank}, b} \leq n]
\]

\[
\text{bank\_pay\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, n, \sigma) = \text{runtime\_dispatch\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, 0, 0, 0, \sigma)
\]

\[
[H = \text{Thief} \land y_{\text{Bank, to\_pay}} \leq n]
\]

\[
\text{pay\_cont\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, r, n, \sigma) = \text{runtime\_dispatch\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}} - n, y_{\text{Thief}}, b, 0, 0, 0, \sigma)
\]

\[
[true]
\]

\[
\text{thief\_fallback}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, \sigma) = \text{bank\_pay\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, 1, \sigma)
\]

\[
[H = \text{Bank}]
\]

\[
\text{runtime\_dispatch\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, r, \sigma) = \text{pay\_cont\_int\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, 0, 0, 0, 1, 6)
\]

\[
s_{1} = \text{Bank} \land s_{2} = \text{pay\_cont\_int\_int}
\]

\[
\text{runtime\_dispatch\_int}(x_{\text{Bank}}, b, x_{\text{Bank, to\_pay}}, x_{\text{Thief}}, b, y_{\text{Bank}}, b, y_{\text{Bank, to\_pay}}, y_{\text{Thief}}, b, H, v, r, \sigma) = \text{max}(0, x_{\text{Bank, b}} - y_{\text{Bank, b}})
\]

\[
[s_{1} = 1]
\]
\[ \downarrow^* = \downarrow, \ldots, \downarrow \]
\[ \sigma = s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_{11}, s_{12} \]
\[ \sigma^h = s_7, s_8, s_9, s_{10}, s_{11}, s_{12}. \]

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. [14], it is possible to remove continuations (of imperative languages) without using any CPS translation. The key to avoid 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 to analyze the cost of the function \( f \) defined in pseudo-C code as follows

\[ f(i)\{ \return g(h(i)) \}. \]

One can use the cost equation

\[ k_f(X,Z) = e + k_f(X,Y) + k_f(Y,Z) \quad |true| \]

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 [23], 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. [23]).

### 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\(^8\): 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, \texttt{push} and \texttt{pop} can then be implemented using large \texttt{if} - \texttt{then} - \texttt{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 to 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 array and finite maps.

<table>
<thead>
<tr>
<th>Example</th>
<th>No CPS number of cost equations</th>
<th>CPS number of cost equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank-Thief code</td>
<td>11</td>
<td>20</td>
</tr>
<tr>
<td>English auction1</td>
<td>29</td>
<td>20</td>
</tr>
<tr>
<td>English auction2</td>
<td>27</td>
<td>38</td>
</tr>
<tr>
<td>Handover Ponzi</td>
<td>1438</td>
<td>336</td>
</tr>
</tbody>
</table>

\(^8\) We are adopting the Solidity terminology: an array is a stack data structure that can grow dynamically, e.g. via push operations.
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 use the operation inside a recursion or an iteration). To test our analyzer, we rewritten the code by replacing the new with a set of pre-instantiated contract name. 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 bachelor thesis [23], 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 address 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 store 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 perform type inference for translating function calls over addresses, since Solidity requires to cast every address to a contract interface.

While the code in Ref. [23] 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 [17], 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. [15,19]. 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 [7]. 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 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 [2]. 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. [2] 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 on 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 [2] is the following one. Computing gas amounts to over-approximate a function GAS(x) (see our Definition 7) and, in Ref. [2], 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. [2]. Finally, up-to our understanding, Albert et al. analyze, analyzes 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 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 RBR mentioned above). One such language is Scilla [24] 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. [12], converts Ethereum Virtual Machine bytecode to semantic logic relations. These relations, paired with the security analysis expressed as logic rules, produces outputs listing potential vulnerabilities. Also Vandal does not model backtrace: 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 behaviours.

Other formal techniques have addressed the critical interplay between smart contracts and users (that are usually untrusted) [8,18,20,21]. In these cases, the model is nondeterministic (because of users’ behaviour) and one tries to predict the maximum profit for some user. 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) behaviours 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 behaviour 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 next
Appendix A Technical details

Table 3

<table>
<thead>
<tr>
<th>Contract</th>
<th>$\parallel$ Smart</th>
<th>$\parallel$ Equations without</th>
<th>$\parallel$ Equations with</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>optimizations</td>
<td>optimizations</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6492</td>
<td>1030</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>98,133</td>
<td>11,825</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1,452,566</td>
<td>170,308</td>
<td></td>
</tr>
</tbody>
</table>

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 is 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 $\text{msg.sender}$ 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 $\text{msg.sender}$ 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 when the smart contracts involved are $2$, $3$ and $4$: the reader may notice that the number of equations decreases by $87\%$ in average. A formal study of (more aggressive) optimizations has not been undertaken so far and is in 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 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. [4,22], using functional languages with expressive types systems, like $\mathsf{F^*}$ [26], or even combining them with amortized analysis in the spirit of Ref. [16] (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 [3]: 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.

Funding

Cosimo Laneve has been partly supported by the H2020-MSCA-RISE project ID 778233 ‘Behavioural Application Progvdam 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’.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Lemma 16. (Substitution Lemma) Let $\Gamma_0, \Gamma_1$ be pure environments and $\Gamma', \Gamma''$ be ground environments. If $\Gamma_0, \Gamma_1; [x] \rightarrow S; \phi \rightarrow \top$ then $\Gamma_0, \Gamma_1; [x] \rightarrow S; \phi \rightarrow \top$. Similarly for expressions.

Proof. Standard induction on the depth of the proof tree of $\Gamma_0, \Gamma_1; [x] \rightarrow S; \phi \rightarrow \top$ and a case analysis on the last rule used.

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

1. (determinism) If $\Theta \Rightarrow \forall \Theta$ then there is at most one $\Theta'$ such that $\Theta \Rightarrow \forall \Theta'$;
2. (correctness) If $S \rightarrow S'$ then there exists $\Theta'$ such that $S \vdash \Theta'$ and $\Theta \Rightarrow \forall \Theta'$.

Proof. Determinism. This follows directly from the translation in Section 4 because, in every $\sum_{i \in 1..n} \phi_i$ $\Theta_i$ at most one $\phi_i$ may be true every time. Correctness. The Proof is by induction on the length of $S \rightarrow S'$. We use the property that, if $\sum_{i \in 1..n} \phi_i \rightarrow e$ and $\{ [e'] \}_{i \in 1..n} = \nu$ where $\\ell$ is the memory of $D$ in some state $S'$ then $e'$ is a ground expression whose value $\{ [e'] \}$ is $\nu$.

The basic case of the induction is immediate; the inductive case $S \rightarrow S' \rightarrow S''$ is demonstrated by means of a case-analysis on the reduction $S \rightarrow S''$.

We discuss only the sub-case when $S \rightarrow S'$ uses rule $\text{METH}$; the other ones are either simpler or similar. Since we are using $\text{METH}$, the following are true:
Additionally, since $S' \vdash \Theta'$ for some $\Theta'$ by induction hypothesis, we have used rule $[\text{INVK-NV}]$ with the hypotheses:

(6) $\Gamma \vdash_{\ell, C_i} e : e_0$

(7) $\Gamma \vdash_{\ell, C_i} \bar{e} : \bar{e}'$

By definition of $\text{env}(S')$ we have $\Gamma'(C_b) = \ell'_b$; therefore, by (2) and (6) we have $e_0 = C_b$ and, by (3) and (7), $\bar{e}'$ are ground expressions whose values are $\bar{v}$.

According to rule $[\text{MEMTH}]$, we have

$$S' = \prod_{i \in \{1, \ell\} \setminus b} C_i(\ell'_i, \ell'_b) | C_b(\ell'_b | \bar{\ell} \rightarrow \bar{v}, \bar{y} \rightarrow \bar{y}, \ell'_b) | C_b : S_a$$

where $\models (\bar{\ell} \bar{x}, \bar{y}) \in C_b$. We demonstrate that there is $\Theta''$ such that $S' \vdash \Theta''$ and $\Theta'' \Rightarrow_\ell \Theta'$. By rule $[\text{FUNCTION}]$,

$$\Gamma_0, \Gamma_1 \vdash_{\bar{\ell}, \bar{v}, \bar{y} \rightarrow \bar{y}}_b \Theta'(\bar{\ell}/\bar{x}) \mid \Gamma_0, \Gamma_1 \rightarrow \Gamma, \Gamma'$$

and, by the Substitution Lemma, we obtain

$$\Gamma, \Gamma' \vdash_{\bar{\ell}, \bar{v}, \bar{y} \rightarrow \bar{y}}_b \Theta'(\bar{\ell}/\bar{x}) \mid \Gamma_0, \Gamma_1 \rightarrow \Gamma, \Gamma'$$

(8)

By definition, (8) is exactly $S' \vdash \Theta'(\bar{\ell}/\bar{x}) \mid \Gamma_0, \Gamma_1 \rightarrow \Gamma, \Gamma'$.

As regards $\Theta'$, we observe that

$$\Theta' = (m \in C_b) C_b(m(\Gamma, \Gamma', \bar{v}, C_i)) + (\text{m.payable} \in C_b) C_b(m(\Gamma, \Gamma', 0, \bar{v}, C_i)) + (m \notin C_b \land \text{m.payable} \notin C_b \land \text{fallback} \in C_b) \Gamma' + (m \notin C_b \land \text{m.payable} \notin C_b \land \text{fallback} \notin C_b) \Gamma$$

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

$$\Theta'' \Rightarrow_\ell \Theta'(\bar{\ell}/\bar{x}) \mid \Gamma_0, \Gamma_1 \rightarrow \Gamma, \Gamma'$$

This concludes the proof.

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

1. **Determinism:** $\text{main}(\langle \ell I \rangle, \bar{v}, \bar{H})$ has a unique $\rightarrow_{\text{CoFloCo}}$-normal form
2. **Correctness:**
   - $\text{main}(\langle \ell I \rangle, \bar{v}, \bar{H}) \rightarrow_{\text{CoFloCo}} \text{GAIN}^C_{\text{CM}}(\Gamma, \bar{v}, \bar{H})$ if we selected $\text{Cost}^C_{\text{gain}}$ during the translation,
   - $\text{main}(\langle \ell I \rangle, \bar{v}, \bar{H}) \rightarrow_{\text{CoFloCo}} \text{LOSS}^C_{\text{CM}}(\Gamma, \bar{v}, \bar{H})$ if we selected $\text{Cost}^C_{\text{loss}}$ during the translation.

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

1. for every ground Presburger expression $e$ whose value is $\bar{v}$, $\bar{v} e \rightarrow_{\text{CoFloCo}} \bar{v}$.

   *Proof:* by inspection of the Definition of $\bar{v} e$.

2. for every ground guard $\phi$, $\phi$ holds if and only if $\bar{v} \phi$ holds.

   *Proof:* by inspection of the Definition of $\bar{v} \phi$, remembering that $\bar{v}$ is injective over functions and contract names.
3. for every code \( \Theta_1 \) if \( \Theta_1 \Rightarrow \Theta_2 \) in the intermediate language and \( \Gamma \Theta_1 \Rightarrow \Theta_1 \) and \( \Gamma \Theta_2 \Rightarrow \Theta_2 \) then \( \Theta_1 \Rightarrow \Theta_2 \) in the simplified intermediate language.

**Proof:** Here we are abusing of the notation because the terms \( \Theta_1 \) and \( \Theta_2 \) are not in the intermediate language but are in the simplified one (which does not use environments at all). Similarly when we write \( \Theta_1 \Rightarrow \Theta_2 \). 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 \( \Rightarrow^* \) over codes, using the previous two points.

4. let \( m([\Gamma_0], \vec{x}) = \sum_{i, j} (\forall_{i, j} \phi_i)(\Theta_i) \) in the simplified intermediate language. Then, for every ground \( \Gamma' \), \( \phi \) and \( n \) such that \( m([\Gamma_0], \vec{x}) \Rightarrow^*_n \Gamma' \) with exactly one derivation, there is exactly one derivation \( m([\Gamma_0], \vec{x}) \Rightarrow \text{CoFloCo} \, \text{cost}([\Gamma_0], \vec{x}) \) (the derivation is deterministic).

**Proof:** by induction on \( n \). Let \( \forall_{i, j} \phi_i \) be the unique guard such that \( (\forall_{i, j} \phi_i)(\{\Gamma_0 \} / \{\Gamma_0 \}) \{\vec{x} / \vec{\tau} \} \). This can happen only if there is at least one \( j \in J^i \) such that \( \phi_i(\{\Gamma_0 \} / \{\Gamma_0 \}) \{\vec{x} / \vec{\tau} \} \). Then

\[
m([\Gamma_0], \vec{x}) \Rightarrow \Theta_i \{\{\Gamma_0 \} / \{\Gamma_0 \}\} \{\vec{x} / \vec{\tau} \}
\]

and

\[
m([\Gamma_0], \vec{x}) \Rightarrow \text{CoFloCo} \, \text{cost}([\Gamma_0], \Theta_i \{\{\Gamma_0 \} / \{\Gamma_0 \}\} \{\vec{x} / \vec{\tau} \})
\]

because of the equation

\[
m([\Gamma_0], \vec{x}) = \text{cost}([\Gamma_0], \Theta_i) \quad [\phi_i]
\]

No other \( \Rightarrow \text{CoFloCo} \)-reduction is possible because:

(a) any other predicate \( \phi_i(\{\Gamma_0 \} / \{\Gamma_0 \}) \{\vec{x} / \vec{\tau} \} \) for \( i \neq j \) and \( j \in J^j \) evaluates to false. This follows by the fact that every other \( (\forall_{i, j} \phi_i)(\{\Gamma_0 \} / \{\Gamma_0 \}) \{\vec{x} / \vec{\tau} \} \) must be false, otherwise the reduction should not have been deterministic on the intermediate terms (obtained translating mSCL programs);
(b) any other disjunct \( \phi_i(\{\Gamma_0 \} / \{\Gamma_0 \}) \{\vec{x} / \vec{\tau} \} \) of \( \forall_{i, j} \phi_i(\{\Gamma_0 \} / \{\Gamma_0 \}) \{\vec{x} / \vec{\tau} \} \) that evaluates to true triggers the same reduction

\[
m([\Gamma_0], \vec{x}) \Rightarrow \text{CoFloCo} \, \text{cost}([\Gamma_0], \Theta_i \{\{\Gamma_0 \} / \{\Gamma_0 \}\} \{\vec{x} / \vec{\tau} \})
\]

because of the equation

\[
m([\Gamma_0], \vec{x}) = \text{cost}([\Gamma_0], \Theta_i) \quad [\phi_i]
\]

By cases over \( n \):

- if \( n = 0 \) then \( \Theta_0(\{\Gamma_0 \} / \{\Gamma_0 \}) \{\vec{x} / \vec{\tau} \} = \Gamma \) and we are done;
- otherwise \( \Theta_0(\{\Gamma_0 \} / \{\Gamma_0 \}) \{\vec{x} / \vec{\tau} \} = m(\{\Gamma_0 \}, \vec{\tau} = \Rightarrow^{-1} \Gamma \) and \( \text{cost}([\Gamma_0], m(\{\Gamma_0 \}, \vec{\tau} = \Rightarrow^{-1} \Gamma \)) = m(\{\Gamma_0 \}, \vec{\tau} = \Rightarrow^{-1} \Gamma \) and we conclude by inductive hypothesis.

5. **Determinism:** main\((\Gamma^1, \Gamma^0, e, \phi, H)\) has a unique \( \Rightarrow \text{CoFloCo} \)-normal-form as a corollary of all previous points and since the new equation main has only one branch.

6. **Correctness:**

- main\((\Gamma^1, \Gamma^0, e, \phi, H)\) if we selected \( \text{Gain}_{\text{Main}}^C \) during the translation,
- main\((\Gamma^1, \Gamma^0, e, \phi, H)\) if we selected \( \text{Loss}_{\text{Main}}^C \) during the translation.

By the previous point, both \( \text{Gain}_{\text{Main}}^C \) and \( \text{Loss}_{\text{Main}}^C \) are defined. Combining the first 5 points, we obtain main\((\Gamma^1, \Gamma^0, e, \phi, H)\)\( \Rightarrow \text{CoFloCo} \, \text{cost}((\Gamma^1, \Gamma^0, e, \phi, H)) \).
References


