Ether Analysis of Solidity functions

Cosimo Laneve and Claudio Sacerdoti Coen

Dept. of Computer Science and Engineering, University of Bologna – INRIA Focus
{cosimo.laneve, claudio.sacerdoticoen}@unibo.it

Abstract. We define a technique to analyze Ether transfers of Solidity smart contracts functions. To this aim, we select a subset of the language and associate cost equations to functions that highlight their input-output behaviours on symbolic representation of the smart contract state. The association is defined by means of an intermediate language that expresses the semantics (of the subset) of Solidity and that can be easily translated into cost equations by taking a cost model expressing Ether transfers. The resulting equations are therefore computed by feeding them to an off-the-shelf cost analyzer. We have also prototyped our technique and we report the initial assessments. We finally discuss the extensions of the prototype to a larger Solidity fragment.

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 programming languages for smart contracts have been recently proposed for specifying decentralized applications, such as the Bitcoin Scripting, which is actually a virtual machine language, for the homonymous blockchain [8], Solidity for Ethereum [10], Move for the Libra blockchain [7]. Security guarantees in these languages are of paramount importance because it is possible to program the transfer of large capitals. In fact, already in the past few years, several millions of USD have been lost because of subtle flaws in the smart contracts [17, 9].

To alleviate the burden of smart contract analysis, a number of automated techniques have been designed for verifying relevant properties, such as liquidity [4], gas consumption [2], and compliance and violation of programming patterns [18]. This contribution follows these lines of research by focussing on another critical feature that is at the core of famous attacks [17, 9]: the transfer of cryptocurrency assets from one address to another. In particular, we propose a technique for automatically compute upper bounds of the amount of cryptocurrency gained or lost by a smart contract during a transaction.

Our programme is as follows. In Section 2, we define μSolidity, a language featuring a minimal set of Solidity smart contract primitives, such as function invocations, field updates, conditional behaviour, cryptocurrency transfer, fallback functions, recursion and failures. Then, in Section 3, we translate μSolidity into an intermediate declarative language that expresses its semantics. The intermediate representation, coupled with the appropriate cost model, is then translated in Section 4 to a set of cost equations that are adequate to off-the-shelf analyzers [11, 1]. We have prototyped our technique and Section 5 reports the preliminary assessments obtained by running the prototype on few archetypal examples (a DAO-like attack, a Ponzi Scheme and an English Auction Scheme). In Section 6 we also study an extension of μSolidity to a larger fragment of Solidity and we deliver our conclusions in Section 7.

Related work. In the past few years formal methods have been largely used to analyze smart contracts with the aim of verifying security properties. Our technique follows the same pattern of previous analyzers proposed by the first author [12, 14]. 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 μSolidity is [5]. 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 techniques yield a precise analysis of conditional statements by restricting the language to guards belonging to Presburger arithmetic (similarly to what we do in this paper).

An interesting paper about asset movements targets Bitcoin Script [4]. 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 μSolidity.

Other formal techniques have addressed the critical interplay between smart contracts and users (that are usually untrusted) [6, 15, 16, 13]. 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.

2 The language MicroSolidity

Micro-Solidity, noted μSolidity, is a subset of the Solidity language featuring a minimal set of smart contract primitives, such as function invocations, field updates, conditional behaviour, cryptocurrency transfer, recursion and failures.

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

The syntax of μSolidity is

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

A μSolidity program is a sequence of smart contract definitions $(\mathsf{C}_1, \cdots, \mathsf{C}_n)$, which, in turn, are sequences of fields and function definitions. In a function definition $\mathsf{function}\ \mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}\ z};\ \mathsf{S}\}$, $\overline{\mathsf{T}\ x}$ are the *formal parameters* and $\overline{\mathsf{T}\ z};\ \mathsf{S}$ is the *body* of $\mathtt{m}$, where $\overline{\mathsf{T}\ z}$ are the local variables. We assume that fields, formal parameters and local variables do not contain duplicate names. Smart contracts have an implicit field, the balance, that records the Ethers stored in the contract. In μSolidity, every function is always `public`, therefore the attribute is omitted; additionally, a function may be `payable` or not. In the first case, function invocations may also carry Ethers (using the keyword value); in the second case no Ether can be transferred. Finally, a smart contract may have the *fallback function* fallback( ) payable { } (there may be at most one such function in a smart contract). If the smart contract has no fallback then Ether transfers to it are refused and a backtrack occurs.

μSolidity also admits invocations of undeclared functions that default to the fallback function, if provided, and backtrack otherwise. The fallback function ignores all actual parameters, except the Ether transferred.

Types T in μSolidity are naturals uint .

Statements S includes the empty statement; the assignment $x = \mathsf{E}$ of x with a continuation S, where x may be either a field or a formal parameter or a local variable; conditionals; the invocation of a function in the two formats $\mathsf{E}.\mathtt{m}(\overline{\mathsf{E}'})$ and $\mathsf{E}.\mathtt{m}.\mathsf{value}(\mathsf{E}'')(\overline{\mathsf{E}'})$ that highlight whether an Ether transfer occurs or not. Statements may also be revert that backtracks the computation to the initial store (this is not completely exact because the consumed gas is never returned, but in this paper we are overlooking this feature), and $\mathsf{E}.\mathsf{transfer}(\mathsf{E}')$

```
1   contract Bank {
2       function() payable { }
3       function pay(uint n) payable {
4           if (msg.value >= 1 && this.balance > n) {
5                       msg.sender.transfer(n) ;
6                       msg.sender.ack() ;
7            }
8       }
9   }
10  contract Thief {
11      function() payable { }
12      function ack() {
13           msg.sender.pay.value(1)(2) ;
14      }
15  }
```

Fig. 1. A DAO-like attack in μSolidity.

that transfers E' Ethers 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 Ethers transmitted 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*.

There are two main differences with Solidity that are taken in order to avoid dependences from the context and augment precision of the cost analysis[1]:

1. we do not address dynamic contract creation and deployment. Therefore 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.
2. μSolidity functions do not return values and fallbacks have empty bodies. This allows us to define a simple translation into an intermediate language without continuations. In Section 6 we report about an extension of the language that weakens this constraint but requires a continuation-passing-like translation.

The features of μSolidity are illustrated by discussing an example.

Example 1. Figure 1 reports the codes of the contracts `Bank` and `Thief`, which are a simplified version of the famous DAO attack [17]. The bank is used for paying clients: it has a balance and, as soon as a client invokes `pay` with an

[1] There are also other differences, which are not very relevant. For example, Solidity has two function invocations: *external* ones, such as $C.\mathtt{m}(\overline{e})$, and *internal* ones, such as $\mathtt{m}(\overline{e})$. The values of msg.sender and msg.value are set only for external invocations; internal invocations keep the corresponding values of the caller. In correspondence, there are a number of modifiers in function declarations to constrain methods to be invoked only internally, or only externally, or in both ways. In μSolidity, function invocations are always external – therefore the pattern $\mathtt{m}(\overline{e})$ is not admitted (in case, you must write this.$m(\overline{e})$) – even if our prototype also covers the internal invocations.

integer `n` and the balance is large enough – line 4 –, it withdraws n Ethers and transfers them to the client – line 5. The function `pay` acknowledges the transaction by invoking client's function `ack` (because the client has payed 1 Ether for it) – line 6. The client `Thief` attacks `Bank` by emptying all its balance. This is performed by the function `ack` that simply invokes back `pay` (by paying at least 1 Ether) and asking to withdraw 2 Ethers (line 13).

Assuming the balance of *Bank* is v and that of *Thief* is 1, the following invocation performed by the *Thief* expresses the attack: `Bank.pay.value(1)(2)`.

3 The translation into the intermediate language

The cost analysis of μSolidity consists of two stages. First, the source program is translated into a declarative language that expresses μSolidity semantics; second, the intermediate code, coupled extended with a cost model expressing the Ether movements, is translated to cost equations understood by the analyzers. The advantage of this approach is that our intermediate code is language independent and can be statically analyzed more straightforwardly than μSolidity source code using multiple techniques, including the translation to cost equations we present in the paper.

The intermediate language. The syntax of intermediate codes Θ is

$$\Theta \ ::= \quad \Gamma \quad | \quad e.m(\Gamma, \Gamma', e', \overline{e''}, H) \quad | \quad \textstyle\sum_{i \in I}(\varphi_i)\ \Theta_i$$

where

- Γ, called *environment*, is a map $(Id \to FId \to \mathsf{E}) \cup (Var \to \mathsf{E})$; we always shorten $\Gamma(C)(\mathtt{f})$ into $\Gamma(C.\mathtt{f})$ and use $\Gamma[C.\mathtt{f} \mapsto e]$ to denote the update to e of the field $C.\mathtt{f}$. Notice that, in general, environments return terms rather than (integer) values.
- φ_i are boolean expressions.

Function calls take in input environments that are therefore used as actual parameters. To express the language semantics we need a corresponding notion of formal parameters, which is the one of a pure environment: an environment is *pure* whenever it is injective and returns only variables.

A *program* in the intermediate language is a tuple $\mathcal{D}$ of function definitions $C.m : (\Gamma_0, \Gamma_1, v, \overline{x}, H) \to \sum_{D \in Id}(H = D)\Theta_D$ (we keep the notation of μSolidity for the name of functions). The formal parameters of a function definition include two environments Γ_0 and Γ_1 that are *pure*, namely they return two disjoint set of variables. The remaining parameters, namely $v, \overline{x}$ and H respectively describe the amount of the transferred Ether, the parameters of the function and the caller name. Below, we also use *ground environments*: an environment is *ground* when the expressions in the codomain are *constant values*.

In order to formalize the semantics of function call, we need to match an actual parameter Γ' that is a ground environment with the formal one Γ that

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

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

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

Fig. 2. Translation of μSolidity expressions

is a pure environments. 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 two rules:

$$\frac{C.m(\Gamma_0, \Gamma_1, x, \overline{z}, H) \to \sum_{D \in Id}(H = D)\Theta_D \in \mathcal{D} \quad [\![e]\!] = u \quad [\![\overline{e'}]\!] = \overline{v}}{C.m(\Gamma, \Gamma', e, \overline{e'}, D) \Longrightarrow_{\mathcal{D}} \Theta_D\{^{u,\overline{v}}/_{x,\overline{z}}\}\sigma_{\Gamma_0,\Gamma}\sigma_{\Gamma_1,\Gamma'}}\ [\textsc{Apply}] \qquad \frac{[\![\varphi_i]\!] = \mathit{true} \quad \Theta_i \Longrightarrow_{\mathcal{D}} \Theta_i'}{\sum_{i \in I}(\varphi_i)\Theta_i \Longrightarrow_{\mathcal{D}} \Theta_i'}\ [\textsc{Choice}]$$

where $[\![e]\!]$ is the value of e. (The definition of $[\![e]\!]$ is omitted because straightforward; as we will see, the expressions may also contain predicates on $\mathcal{D}$, such as $m \in C$ or $m.\mathit{payable} \in C$.) We notice that the semantics of Θ is *nondeterministic*: $(1 > 0)\Theta_1 + (2 > 1)\Theta_2$ may evolve into either Θ_1 or Θ_2.

The translation of μSolidity. We define the translation of μSolidity in the intermediate language by using judgments and inference rules. The judgments have the following form:

- judgments for expressions $\Gamma \vdash^{e}_{C,D} \mathsf{E} : e'$, where e and e' are expressions that contain constants or variables, C and D are respectively the caller and the callee contracts;
- judgments for statements $\Gamma, \Gamma' \vdash^{e}_{C,D} \mathsf{S} : \Theta$, where Γ is the *committed* environment, Γ' is the *current* environment and Θ is the *resulting* intermediate code.

The translation of expressions is reported in Figure 2. This translation partially evaluates expressions by replacing accesses to fields with the corresponding values in the environment. The resulting intermediate codes will be used by our cost analysis in Section 4. Rules [FIELD] and [VAR] manage variables; there are three cases: a variable is a callee's field, or it is a formal parameter or a smart contract name. In any case we return the corresponding value in Γ. The function $\mathit{fields}(\Gamma(C))$ returns $\mathit{dom}(\Gamma(C))$. In [BAL] the translation of $e_0.\mathsf{balance}$ is the balance of a contract; in this case it is necessary that e_0 is a smart contract name: in our setting we write $e_0 \in Id$. Rule [PROD-DIV] addresses multiplication

$$\frac{}{\Gamma, \Gamma' \vdash^{e}_{C,D} \varepsilon : \Gamma'}\ [\textsc{empty}] \qquad \frac{}{\Gamma, \Gamma' \vdash^{e}_{C,D} \mathsf{revert}\,; : \Gamma}\ [\textsc{revert}]$$

$$\frac{x \in \mathit{fields}(\Gamma(D))\backslash\{\mathsf{balance}\} \quad \Gamma' \vdash^{e}_{C,D} \mathsf{E} : e' \quad \Gamma, \Gamma'[D.x \mapsto e'] \vdash^{e}_{C,D} \mathsf{S} : \Theta}{\Gamma, \Gamma' \vdash^{e}_{C,D} x = \mathsf{E};\mathsf{S} : \Theta}\ [\textsc{asgn}]$$

$$\frac{x \notin \mathit{fields}(\Gamma(D)) \quad \Gamma' \vdash^{e}_{C,D} \mathsf{E} : e' \quad \Gamma, \Gamma'[x \mapsto e'] \vdash^{e}_{C,D} \mathsf{S} : \Theta}{\Gamma, \Gamma' \vdash^{e}_{C,D} x = \mathsf{E};\mathsf{S} : \Theta}\ [\textsc{asgn-var}]$$

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

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

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

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

Fig. 3. The translation of μSolidity statements

and division. Since the cost analysis of Section 4 only covers Presburger arithmetics expressions where the second argument of products and divisions are constants, the inference rules do not translate expressions that cannot be fed to the analyzer.

The translation of statements is defined in Figure 3. The judgments return intermediate codes that use the predicates

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

We also use two update operations on environments: $\Gamma[C.\mathtt{f} \mapsto^{+} e] \stackrel{\text{def}}{=} \Gamma[C.\mathtt{f} \mapsto \Gamma(C.\mathtt{f}) + e]$ and $\Gamma[C.\mathtt{f} \mapsto^{-} e] \stackrel{\text{def}}{=} \Gamma[C.\mathtt{f} \mapsto \Gamma(C.\mathtt{f}) - e]$.

Rules [INVK-NV] and [INVK] define function invocations for non-payable functions and payable ones, respectively. The former one returns a choice between several alternatives: (i) when m is in e_0 and it reduces to the invocation; (ii) when m is in e_0 and it is payable then it is translated to the invocation with

$$\text{[FUNCTION]}\quad \frac{\left(\begin{array}{l}\Gamma_0(D) = [\mathtt{f}_1 \mapsto x_{D,1}, \cdots, \mathtt{f}_n \mapsto x_{D,n}, \mathsf{balance} \mapsto x_{D,b}] \\ \Gamma_1(D) = [\mathtt{f}_1 \mapsto y_{D,1}, \cdots, \mathtt{f}_n \mapsto y_{D,n}, \mathsf{balance} \mapsto y_{D,b}]\end{array}\right)^{D \in Id} \quad \left(\Gamma_0, \Gamma_1[\overline{x} \mapsto \overline{x_0}, \overline{y} \mapsto \overline{0}] \vdash^{v}_{C,D} \mathsf{S} : \Theta_C\right)^{C \in Id}}{\mathsf{function}\ \mathtt{m}(\overline{\mathsf{T}\ x})[\mathsf{payable}]\{\overline{\mathsf{T}\ y};\ \mathsf{S}\} \in D}$$

$$\Gamma_0, \Gamma_1 \vdash D.m : (\Gamma_0, \Gamma_1, v, \overline{x_0}, H) \to \textstyle\sum_{C \in Id}(H = C)\ \Theta_C$$

$$\text{[PROGRAM]}\quad \frac{\mathcal{D} = \left(\Gamma_0, \Gamma_1 \vdash D.m : (\Gamma_0, \Gamma_1, v, \overline{x}, H) \to \Theta_{D.m}\right)^{D.m \in \mathsf{D}_i, i \in 1..n}}{\vdash \left(\mathsf{D}_1, \cdots, \mathsf{D}_n\right) : \mathcal{D}}$$

Fig. 4. The translation for μSolidity functions and programs

0 Ether transferred; (*iii*) when m is not in e_0 but the contract has the fallback function then the translation is the call to fallback that, in our case, returns the current environment (because fallback has empty body); (*iv*) when both m and fallback are not in e_0 then a backtrack occurs and the translation is the backtrack environment. Rule [INVK] manages invocations with Ether transfer from the caller to the callee; in this case we must check that the caller has enough Ether in his balance, otherwise a backtrack occurs.

The translation of μSolidity is completed with the rules for function definition and programs, given in Figure 4. We write $\mathsf{function}\ \mathtt{m}(\overline{\mathsf{T}\ x})[\mathsf{payable}]\{\mathsf{S}\} \in D$ to mean that $\mathtt{m}$ is declared in the smart contract D whose address is D. In [FUNCTION], the definition of a function is given in two pure environments that act as formal parameters. The critical point is that, in our system, the set Id is *finite*, therefore the hypotheses of rule [FUNCTION] and the choice in the conclusion are finite. (Said otherwise, we analyze the cost of *smart contract programs with a finite number of known contract instances.*) Rule [PROGRAM] gives the translation of a smart contract program.

Example 2. To illustrate the translation defined in this section, we apply it to the functions of Example 1. For *Bank.pay* we obtain:

$$Bank.pay : (\Gamma_0, \Gamma_1, v, n, H) \to \textstyle\sum_{D \in Id}(H = D)\big[\ (v \geqslant 1\ \wedge\ y_{Bank,b} \geqslant n)\ \Theta \\ +!(v \geqslant 1\ \wedge\ y_{Bank,b} \geqslant n)\ \Gamma_1\ \big]$$

where $\Theta = (y_{Bank,b} \geqslant n\ \wedge\ fallback \in D)\ \Theta' + (y_{Bank,b} < n)\ \Gamma_0 + (fallback \notin D)\ \Gamma_0$

$$\begin{array}{rl}\Theta' = & (ack \in D)\ D.ack(\Gamma_0, \Gamma_1', 0, Bank) \\ + & (ack.payable \in D)\ D.ack(\Gamma_0, \Gamma_1', 0, Bank) \\ + & (ack \notin D\ \wedge\ ack.payable \notin D\ \wedge\ fallback \in D)\ \Gamma_1' \\ + & (ack \notin D\ \wedge\ ack.payable \notin D\ \wedge\ fallback \notin D)\ \Gamma_0\end{array}$$

$$\Gamma_1' = \Gamma_1[Bank.balance \mapsto^{-} n, D.\mathsf{balance} \mapsto^{+} n]$$

For *Thief.ack* we get:

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

where $\Theta = (pay \in D)\ \Gamma_0$
$+\ \ (pay.payable \in D\ \wedge\ y_{Thief,b} \geqslant 1)\ \Theta'$
$+\ \ (pay.payable \in D\ \wedge\ y_{Thief,b} < 1))\ \Gamma_0$
$+\ \ (pay \notin D\ \wedge\ pay.payable \notin D\ \wedge\ fallback \in D\ \wedge\ y_{Thief,b} \geqslant 1))\ \Gamma_1'$
$+\ \ (pay \notin D\ \wedge\ pay.payable \notin D\ \wedge\ fallback \in D\ \wedge\ y_{Thief,b} < 1))\ \Gamma_0$
$+\ \ (pay \notin D\ \wedge\ pay.payable \notin D\ \wedge\ fallback \notin D)\ \Gamma_0\ ]$

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

The correctness of the translation is stated below [2].

Theorem 1. *Let* $(\mathsf{C}_1, \cdots, \mathsf{C}_n)$ *be a* μSolidity *program and let* $\vdash (\mathsf{C}_1, \cdots, \mathsf{C}_n) : \mathcal{D}$. *Let also* $\Gamma, \Gamma' \vdash^{v}_{C,D} \mathsf{S} : \Theta$, *where* Γ *and* Γ' *are ground environments. Then*

1. *(*determinism*) If* $\Theta \Longrightarrow^{*}_{\mathcal{D}} \Theta'$ *then there is at most one* Θ'' *such that* $\Theta' \Longrightarrow_{\mathcal{D}} \Theta''$;
2. *(*correctness*) [Informal statement] Every state reachable from the initial state whose statement is* S *and whose data is given by* Γ, Γ' *has a translation, say* Θ', *and* $\Theta \Longrightarrow^{*}_{\mathcal{D}} \Theta'$.

4 The Ether analysis

The cost model of μSolidity. The intermediate code programs generated by the translation in Section 3 return environments (when they terminate). This output is too informative in our case since we are interested in computing Ether movements of *exactly one critical* smart contract, which are stored in the `balance` field. Additionally, we are not interested in the final value of the `balance`, but in the Ethers that the smart contract can either *loose* or *gain* during a transaction.

Therefore, for terminating programs, we have two cost models: one returns the *maximal gain* of Ethers for a given smart contract, obtained by the difference between its final and initial balances, the other one returns the *maximal lost* of Ethers, which is the converse difference. In order to apply these models to our intermediate terms we perform the following operations (defined according to the structure of the terms):

- if the term is Γ, we derive the cost computing the difference discussed above;
- if the term is either an invocation or a deterministic choice, the costs is 0 (plus the cost of the sub-programs).

It is worth to notice that reverting to the initial state yields the cost 0 and that infinite computations, which are meaningless for smart contracts because computations eventually abort by gas exhaustion, have also cost 0.

[2] The statement is in part informal (e.g. *state reachability in* μSolidity) because we are missing the formal semantics of μSolidity in this extended abstract. The reviewer is referred to the full paper at `cs.unibo.it/~laneve/papers/EtherAnalysisFull.pdf`.

The syntax of `CoFloCo`. In our setting, a *transaction* is the evaluation of a function invocation where the current (e.g. the initial) and committed environments coincide (we might also consider other types of statements). This evaluation is by a set of cost equations (that are adequate to a solver); in turn, each equation describes a step of computation and associates to it its cost according to the cost model described above. The solver we will use is `CoFloCo` [11], even if our outputs also comply with `PUBS` [1].

Solvers take a list of equations in input that are terms [11]

$$m(\overline{x}) \;=\; e + \sum_{i \in 1..n} m_i(\overline{e_i}) \qquad\qquad [\,\varphi\,]$$

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

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

$$e \;::=\; \quad q \;\mid\; e+e \;\mid\; e-e \;\mid\; q*e \;\mid\; max(e_1, \cdots, e_k)$$

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

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

The translation. To define in detail the translation, we will use the following operations:

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

$$\text{let} \quad \Gamma = \Big[\; C_1 \mapsto [\mathtt{f}_{1,1} \mapsto e_{1,1}, \cdots, \mathtt{f}_{1,n_1} \mapsto e_{1,n_1}, \mathsf{balance} \mapsto e_{1,b}], \\ \cdots, C_k \mapsto [\mathtt{f}_{k,1} \mapsto e_{k,1}, \cdots, \mathtt{f}_{k,n_k} \mapsto e_{k,n_k}, \mathsf{balance} \mapsto e_{k,b}] \;\Big]$$

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

$$\lfloor \Gamma \rceil \stackrel{\text{def}}{=} \Big(e_{1,1}, \cdots, e_{1,n_1}, e_{1,b}, \cdots, e_{k,1}, \cdots, e_{k,n_k}, e_{k,b}\Big)$$

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

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

where, for every function name m, the term "$m \in \mathit{fun}(D)$" is true if m is a non payable function declared in D; "$m.\mathtt{p} \in \mathit{fun}(D)$" is true if m is a payable function declared in D. The encoding of D, $D.m$, $D.m.\mathtt{p}$ can be picked to be any injective function.
- the encoding $\ulcorner \cdot \urcorner$ is lifted to the intermediate code as follows:

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

In order to define the set of equations, we need to rewrite the intermediate code in a *canonical form*. As a first step we put every formula ϕ in disjunctive normal form plus the additional constraint that atomic formulae are inequalities. For example, $e \neq e'$ becomes $e < e' \vee e' < e$. Then we derive cost equations by the corresponding intermediate program as follows: for every function $C.m : (\Gamma_0, \Gamma_1, v, \overline{x}, H) \to \sum_{D \in Id} (H = D)\Theta_D$, let $\bigvee_{i \in 1..h} (\varphi_i)\ \Theta_i$ be the canonical form of $\ulcorner \Theta_D \urcorner$ (therefore every φ_i is a conjunction). We have the following cost equations:

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

where

- cost is either $\mathit{cost}^{C_i}_{\mathit{gain}}$ or $\mathit{cost}^{C_i}_{\mathit{loss}}$ for some C_i where we use $\mathit{cost}^{C_i}_{\mathit{gain}}(\lfloor \Gamma_0 \rfloor, \lfloor \Gamma \rfloor) = \Gamma(C_i.\mathit{balance}) - \Gamma_0(C_i.\mathit{balance})$ to compute the upper bound for the gain of contract C_i, and $\mathit{cost}^{C_i}_{\mathit{loss}}(\lfloor \Gamma_0 \rfloor, \lfloor \Gamma \rfloor) = \Gamma_0(C_i.\mathit{balance}) - \Gamma(C_i.\mathit{balance})$ to compute the upper bound for its loss.
- $\mathit{cost}(\lfloor \Gamma_0 \rfloor, e.m(\lfloor \Gamma_0 \rfloor, \lfloor \Gamma \rfloor, \overline{e'})) = e.m(\lfloor \Gamma_0 \rfloor, \lfloor \Gamma \rfloor, \overline{e'})$

Finally, if we are interested in the analysis of a transaction that invokes the method $e.m$, we add a first equation

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

where $b_1, \ldots, b_n$ are the variables in $\lfloor \Gamma \rfloor, \overline{y}$ of type uint . Assuming them to be non negative is required because variables in `CoFloCo` are assumed to be signed.

Example 3. To illustrate the output of our technique we compute the cost equations of the functions `Bank.pay` and `Thief.ack` in Example 2, according to the cost model that computes the loss of the `Bank`. We shorten `Bank` and `Thief` into B and T, respectively; we write $[\mathit{fallback} \in T \ \wedge \ \mathit{ack} \in T]$ for readability sake, even if it is not produced by the translator because true. Similarly for the other predicates of the same shape.

$$\begin{array}{l} \mathit{main}(x_{B,b}, x_{T,b}, v, n, H) = B.\mathit{pay}(x_{B,b}, x_{T,b}, x_{B,b}, x_{T,b}, v, n, H) \\ \qquad\qquad [x_{B,b} \geqslant 0 \wedge x_{T,b} \geqslant 0 \wedge v \geqslant 0 \wedge n \geqslant 0] \\ B.\mathit{pay}(x_{B,b}, x_{T,b}, y_{B,b}, y_{T,b}, v, n, H) = T.\mathit{ack}(x_{B,b}, x_{T,b}, y_{B,b} - n, y_{T,b} + n, 0, B) \\ \qquad\qquad [H = T \ \wedge \ v \geqslant 1 \ \wedge \ y_{B,b} \geqslant n \ \wedge \ \mathit{fallback} \in T \ \wedge \ \mathit{ack} \in T] \\ B.\mathit{pay}(x_{B,b}, x_{T,b}, y_{B,b}, y_{T,b}, v, n, H) = x_{B,b} - y_{B,b} \qquad [H = T \ \wedge \ v < 1] \\ B.\mathit{pay}(x_{B,b}, x_{T,b}, y_{B,b}, y_{T,b}, v, n, H) = x_{B,b} - y_{B,b} \qquad [H = T \ \wedge \ y_{B,b} < n] \\ B.\mathit{pay}(x_{B,b}, x_{T,b}, y_{B,b}, y_{T,b}, v, n, H) = x_{B,b} - (y_{B,b} - n + n) \\ \qquad [H = B \ \wedge \ v \geqslant 1 \ \wedge \ y_{B,b} \geqslant n \ \wedge \ \mathit{fallback} \in B \ \wedge \ \mathit{ack} \notin B \ \wedge \ \mathit{ack.payable} \notin B] \\ B.\mathit{pay}(x_{B,b}, x_{T,b}, y_{B,b}, y_{T,b}, v, n, H) = x_{B,b} - y_{B,b} \qquad [H = B \ \wedge \ v < 1] \end{array}$$

	♯ LOC	♯ LMC	♯ Equations	`CoFloCo`'s Time for gain + loss
Simplified DAO Attack	20	20	20	734ms + 240ms
Handover Ponzi Scheme	42	50	336	6,964ms + 4,784ms
English Auction Scheme	32	32	38	509ms + 468ms

Table 1. Statistics on a few archetypal examples

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

In the next section we discuss `CoFloCo` [11] outputs when these equations are fed to the tool.

5 Preliminary assessments

We prototyped the cost analyzer in about 2,500 lines of OCaml code. The code is then compiled to `JavaScript` to be run in the browser and can be found at the address: `https://sacerdot.github.io/SmartAnalysis/behavioral_types`. Our tool takes in input a list of smart contract declarations, produces a list of cost equations, and computes the cost equations of the first function of the first contract, say C. The user can choose between two cost models: the *maximum gain* of C's balance and the *maximum loss* of C's balance. 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 is a function of the input parameters of the analyzed function, the initial value of all contract fields, including balances, and every possible invoker of the method.

The number of cost equations returned by our prototype is bi-linear in the number of methods and the number of contracts when the only variable of type `address` is `msg.sender`. We notice that, in the following examples, we have also used the extension of the prototype that also deals with `address` data types (see Section 6), which makes the number of cost equations exponential with respect to `address` variables, where the base is the number of contracts.

To test the tool we have identified three archetypal contracts and, to gain preliminary experience, we rewrote the Solidity code in μSolidity. This required little programming overhead: we had to modify function calls with explicit continuations by moving the continuations inside method bodies.

In Table 1, for every program, we give the lines of of original code (♯ LOC), those produced by our translation (♯ LMC), the number of equations produced (♯ Equations) and the sum of `CoFloCo` times for computing the maximum gain and

the maximum loss. Few remarks about the output of the analyzer are in order. In the simplified DAO attack of Figure 1, the costs are a function of the initial values of `Bank`'s balance (`Bank__balance_`), `Thief`'s balance (`Thief__balance_`), the invoker of the analyzed method (`_msg_sender_`), the amount of coins passed to the method (`_msg_value_`) and the method parameter (`N`). `CoFloCo`'s output is:

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

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

The output shows that the attack can be successful: the bank can loose 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 less than two coins. A careful analysis by hand of the code tells us that the maximum loss bound is strict, while the maximum gain bound is not: the bank can actually only win one coin.

In the "Handover Ponzi scheme" of [3], 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 analyse two scenarios. The first scenario is when `Player` joins the scheme, followed by `Player2`. The analysis yields:

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

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

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

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

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

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

In this scenario `Player` looses all the money he invested.

In the English Auction Scheme, the smart contract Auctioneer records the address of the better that is currently winning the auction, together with his bet. When a new bet arrives, if it is greater than the currently winning one, the previous winner is refunded. Otherwise the bet is refunded to the sender.

The result of the analysis is interesting:

```
MAXIMUM GAIN:
Maximum cost of main__(Better1__balance_,Auctioneer__balance_,Auctioneer_max,
     Bet1, ...): 0
Asymptotic class: constant

MAXIMUM LOSS:
Maximum cost of main__(Better1__balance_,Auctioneer__balance_,Auctioneer_max,
     Bet1, ...): max([nat(Bet1),nat(-Auctioneer_max+Bet1)])
Asymptotic class: n
```

The better cannot gain any money by betting: either he can loose all its bet `nat(Bet1)` (because he is winning the auction) or it can loose the lesser amount `nat(-Auctioneer_max+Bet1)` because he was already winning and decided to lift his offer (the previous offer is returned back).

6 Extensions of the analysis to Solidity

Two relevant features of Solidity that are included in our analyzer and have not yet been discussed are (*i*) data types other than integers (signed integers, booleans, and addresses) and (*ii*) functions returning values and function calls having continuations.

As regards (*i*), we discuss the extension with `address` (the other ones are simpler). Since we use a finite set of symbolic names for contracts, the translation into our intermediate code must consider all the possible instances of the addresses (which are finitely many). For example, the translation rule for functions whose formal parameters are also addresses becomes

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

(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.

Dealing with (*ii*) is not straightforward because our intermediate language in Section 3 only admits tail recursive (or tail mutual recursive) invocations, that are later translated to cost equations. (Even if cost equations analyzers also cover function calls with continuations, we have preferred to stick to the tail call restriction in order to be free of using different sets of tools in the future.) It is worth to notice that an analyzer that also deals with explicit continuations allows one to verify fallback functions with nonempty bodies.

Our extension of the translation in Section 3 for removing continuations uses the standard CPS translation, with one difference. Instead of using a higher order language (which is required by CPS translations), we keep the same intermediate language by extending functions' arguments with an additional one representing the stack of activation records for continuations. In order to keep the code simple, all frames have the same size, which is obtained by computing the maximum number of local parameters of functions, say ι. Then we compute the graph of invocations and verify that it is acyclic, i.e. we give weight 0 to tail invocations and 1 to the other invocations; the graph must have paths of finite weight, otherwise the program is rejected. If κ is the maximal weight of a path in the graph of invocations, the size of stack of frames is bounded by $\iota \times \kappa$.

7 Conclusions

In this paper we have analyzed Ether movements of smart contract functions written in a sublanguage of Solidity. Our technique consists of expressing the semantics of Solidity with an intermediate language that is amenable to be analyzed via cost equations once a cost model has been defined. The cost equations can be fed to a cost analyzer, which allows us to obtain cost values with respect to the inputs. The present contribution is our initial work on this topic and it is in fact a proof-of-concept that demonstrates the adequacy of our technique. We have already delivered correctness proofs of the overall technique and a preliminary assessment of our tool with respect to relevant examples.

Several extensions need to be investigated in the next future. They mostly concern our analyzer. The current extension of the tool with the `address` data type gives a number of cost equations that is exponential with respect to `address` variables. This either downgrades the performance of `CoFloCo` or makes it fail. For example, we tried to encode another Ponzi scheme from [3] and we produced over 300,000 equations for only four actors. In order to reduce the number of equations, we will study optimizations of our translation to perform the disjoint sum over all possible contracts lazily. Indeed, experiments by hand show that the current translation tends to produce sets of equations that differ only over irrelevant details (e.g. a `msg.sender` which is later never used anyway).

The extension of our tool to cover explicit continuations does not yield a larger number of cost equations. However, both `CoFloCo` and `PUBS` fail in computing the costs of equations we generate even on simple examples. In private communication with Samir Genaim, we discovered that minor changes to the tools could be sufficient to make them effective for us. We plan a future collaboration to overcome the problem.

We also plan to extend the analysis to cover vectors and maps having a finite domain, like addresses. Finally, we want to analyze our intermediate code using techniques and tools that are different from cost equations solvers, such as [5].

References

1. Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. Closed-form upper bounds in static cost analysis. *Journal of Automated Reasoning*, 46(2):161–203, Feb 2011.
2. Elvira Albert, Jesús Correas, Pablo Gordillo, Guillermo Román-Díez, and Albert Rubio. SAFEVM: a safety verifier for ethereum smart contracts. In *Proc. of Software Testing and Analysis, ISSTA'19*, pages 386–389. ACM, 2019.
3. Massimo Bartoletti, Salvatore Carta, Tiziana Cimoli, and Roberto Saia. Dissecting Ponzi schemes on Ethereum: Identification, analysis, and impact. *Future Gener. Comput. Syst.*, 102:259–277, 2020.
4. Massimo Bartoletti and Roberto Zunino. Verifying liquidity of bitcoin contracts. In Flemming Nielson and David Sands, editors, *Principles of Security and Trust*, pages 222–247. Springer International Publishing, 2019.
5. Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, et al. Formal verification of smart contracts: Short paper. In *Proc. of Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.
6. Giancarlo Bigi, Andrea Bracciali, Giovanni Meacci, and Emilio Tuosto. Validation of decentralised smart contracts through game theory and formal methods. In *Programming Languages with Applications to Biology and Security*, volume 9465 of *Lecture Notes in Computer Science*, pages 142–161. Springer, 2015.
7. Sam Blackshear and et. Al. Move: A language with programmable resources. Available at https://developers.libra.org/docs/assets/papers/ libra-move-a-language-with-programmable-resources.pdf, 2019.
8. Harris Brakmić. *Bitcoin Script*, pages 201–224. Apress, Berkeley, CA, 2019.
9. Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gun Sirer. An in-depth look at the parity multisig bug. Available at http://hackingdistributed.com/2017 /07/22/deep-dive-parity-bug/, 2017.
10. Chris Dannen. *Introducing Ethereum and Solidity: Foundations of Cryptocurrency and Blockchain Programming for Beginners*. Apress, Berkely, USA, 2017.
11. Antonio Flores Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In *Proceedings of 12th Asian Symposium on Programming Languages and Systems*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, 2014.
12. Abel Garcia, Cosimo Laneve, and Michael Lienhardt. Static analysis of cloud elasticity. *Sci. Comput. Program.*, 147:27–53, 2017.
13. Cosimo Laneve, Claudio Sacerdoti Coen, and Adele Veschetti. On the prediction of smart contracts' behaviours. In *From Software Engineering to Formal Methods and Tools, and Back - Essays Dedicated to Stefania Gnesi on the Occasion of Her 65th Birthday*, volume 11865 of *Lecture Notes in Computer Science*, pages 397–415. Springer, 2019.
14. Cosimo Laneve, Michael Lienhardt, Ka I Pun, and Guillermo Román-Díez. Time analysis of actor programs. *J. Log. Algebr. Meth. Program.*, 105:1–27, 2019.

15. Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proc. of the Conference on Computer and Communications Security*, pages 254–269. ACM, 2016.
16. Bernhard Mueller. Smashing Ethereum smart contracts for fun and real profit. *HITB SECCONF Amsterdam*, 2018.
17. David Siegel. Understanding the dao attack. *Retrieved June*, 13:2018, 2016.
18. Petar Tsankov, Andrei Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin Vechev. Securify: Practical security analysis of smart contracts. In *Proc. of Computer and Communications Security*, CCS '18, pages 67–82. ACM, 2018.

A Technical details

A.1 Semantics of μSolidity programs

The following auxiliary functions are used in the semantic rules:

- $\ell[\mathtt{f} \mapsto v]$ is the *memory update*, namely $(\ell[\mathtt{f} \mapsto v])(\mathtt{f}) = v$ and $(\ell[\mathtt{f} \mapsto v])(\mathtt{g}) = \ell(\mathtt{g})$, when $\mathtt{g} \neq \mathtt{f}$.
- $[\![e]\!]_{C',v,C,\ell}$ is a partial function that returns the value of e assuming C be the current contract, v be the value that has been transmitted during the invocation, C' be the caller and ℓ be the memory of C where values of fields and variables occurred in e are stored. We omit the definition of $[\![e]\!]_{C',v,C,\ell}$, but we require that it never fails. This follows by the μSolidity constraint that, in a division, the second argument is always a non null constant. $[\![\overline{e}]\!]_{C',v,C,\ell}$ returns the tuple of values of $\overline{e}$.

A *state* of a μSolidity program $\left(\mathsf{C}_1, \cdots, \mathsf{C}_n\right)$, ranged over by $\mathcal{S}$, $\mathcal{S}'$, $\cdots$, is a *parallel composition* of (runtime) *contracts* $C(\ell \cdot \ell')$ and *exactly one runtime statement*. Contracts $C(\ell \cdot \ell')$ have a *unique* name C, which is one of the smart contract names of the program, e.g. $\mathit{typeof}(C_i) = \mathsf{C}_i$ for $1 \leqslant i \leqslant n$, and *pairs of memories* $\ell \cdot \ell'$ where ℓ is the *current memory* and ℓ' is the *last committed memory*. Runtime statements may be either $\mathtt{0}$, the terminated statement, or $C, v, D : \mathsf{S}$, where S must be evaluated into the contract D, with a caller C and with a value v. Figure 5 reports the syntax of states. In case of commits, the current memory ℓ of runtime contracts becomes the last committed memory; in case of failures, the system will backtrack by restoring ℓ'. As usual, parallel composition in states is associative and commutative.

The semantics of μSolidity programs is defined by means of transition relation $\mathcal{S} \xrightarrow{\mu} \mathcal{S}'$, where μ may be either empty or ✓ or fail. The program is kept implicit in the transition relation. The reader may find the formal definition of $\xrightarrow{\mu}$ in Figure 5. Let us comment some semantic rules. Rule [UPD] defines the semantics of an update of a field or a variable: the expression e is evaluated in an contract C; the resulting memory binds the value to x. Rules [TRANSFER] and [TRANSFER-FAIL] define the semantics of $e.\mathsf{transfer}(e')$. The first rule verifies that the recipient e 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 (because fallbacks, if any, are empty in μSolidity). The second rule deals with errors: either the recipient is not payable or the caller has not enough balance. In this case a failure occurs and it is propagated to the whole solution (with rule [BKT]).

The last six rules of Figure 5 deal with method invocations, which are particularly complex in μSolidity. Rule [METH] defines successful non-payable method invocations $C.\mathtt{f} = e.\mathtt{m}(\overline{e'})\mathtt{;}s$. In this case, the method dispatch is performed by using the value C'' of e and the statement to evaluate becomes the body of $\mathtt{m}$ (without any continuation). Rules [METH-FB] and [METH-ERR] define unsuccessful non-payable method invocations. The two rules deal with the two subcases whether the callee has a fallback method or not; in the first one, the

States:

$$\mathcal{S} \;::=\; \prod_{i\in I} C_i(\ell_i \cdot \ell_i') \mid C', v, C : \mathsf{S} \qquad \mid \qquad \prod_{i\in I} C_i(\ell_i \cdot \ell_i') \mid 0 \qquad \text{(states)}$$

State transition $\xrightarrow{\mu}$ (μ may be empty, $\checkmark$ or fail): $[\![e]\!]_{C',v,C,\ell}$ never fails]

$$\frac{}{C', v, C : \varepsilon \xrightarrow{\checkmark} 0}\ [\textsc{cmt}] \qquad \frac{}{C', v, C : \mathsf{revert} \xrightarrow{\mathsf{fail}} 0}\ [\textsc{revert}]$$

$$\frac{[\![e]\!]_{C',v,C,\ell} = v'}{C(\ell \cdot \ell') \mid C', v, C : x = e; \mathsf{S} \longrightarrow C(\ell[x \mapsto v'] \cdot \ell') \mid C', v, C : \mathsf{S}}\ [\textsc{upd}]$$

$$\frac{[\![e]\!]_{C',v,C,\ell} \neq 0}{C(\ell \cdot \ell') \mid C', v, C : \mathsf{if}\ (e)\ \{\ \mathsf{S}\ \}\ \mathsf{else}\ \{\ \mathsf{S}'\ \} \longrightarrow C(\ell \cdot \ell') \mid C', v, C : \mathsf{S}}\ [\textsc{if-true}]$$

$$\frac{[\![e]\!]_{C',v,C,\ell} = 0}{C(\ell \cdot \ell') \mid C', v, C : \mathsf{if}\ (e)\ \{\ \mathsf{S}\ \}\ \mathsf{else}\ \{\ \mathsf{S}'\ \} \longrightarrow C(\ell \cdot \ell') \mid C', v, C : \mathsf{S}'}\ [\textsc{if-false}]$$

$$\frac{\begin{array}{c}[\![e]\!]_{C',v,C,\ell} = C'' \quad [\![e']\!]_{C',v,C,\ell} = v' \\ \ell(\mathsf{balance}) \geqslant v' \quad \mathsf{fallback} \in \mathit{typeof}(C'') \\ \ell_1 = \ell[\mathsf{balance} \mapsto^- v] \quad \ell_2 = \ell''[\mathsf{balance} \mapsto^+ v]\end{array}}{\begin{array}{c}C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C', v, C : e.\mathsf{transfer}(e');\ \mathsf{S} \\ \longrightarrow C(\ell_1 \cdot \ell') \mid C''(\ell_2 \cdot \ell''') \mid C', v, C : \mathsf{S}\end{array}}\ [\textsc{transfer}]$$

$$\frac{\begin{array}{c}[\![e]\!]_{C',v,C,\ell} = C'' \quad [\![e']\!]_{C',v,C,\ell} = v' \\ \Big(\ell(\mathsf{balance}) < v' \quad \text{or} \quad \mathsf{fallback} \notin \mathit{typeof}(C'')\Big)\end{array}}{\begin{array}{c}C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C', v, C : e.\mathsf{transfer}(e');\ \mathsf{S} \\ \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid C''(\ell''' \cdot \ell''') \mid 0\end{array}}\ [\textsc{transfer-fail}]$$

$$\frac{\mathcal{S} \xrightarrow{\checkmark} \mathcal{S}'}{C(\ell \cdot \ell') \mid \mathcal{S} \xrightarrow{\checkmark} C(\ell \cdot \ell) \mid \mathcal{S}'}\ [\textsc{cmt}] \qquad \frac{\mathcal{S} \xrightarrow{\mathsf{fail}} \mathcal{S}'}{C(\ell \cdot \ell') \mid \mathcal{S} \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid \mathcal{S}'}\ [\textsc{bkt}] \qquad \frac{\mathcal{S} \longrightarrow \mathcal{S}'}{C(\ell \cdot \ell') \mid \mathcal{S} \longrightarrow C(\ell \cdot \ell') \mid \mathcal{S}'}\ [\textsc{tau}]$$

$$\frac{\begin{array}{c}[\![e]\!]_{C',v,C,\ell} = C'' \quad [\![\overline{e'}]\!]_{C',v,C,\ell} = \overline{v'} \\ \mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ z;\mathsf{S}_\mathtt{m}}\} \in \mathit{typeof}(C'')\end{array}}{\begin{array}{c}C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C', v, C : e.\mathtt{m}(\overline{e'}) \\ \longrightarrow C(\ell \cdot \ell') \mid C''(\ell''[\overline{x} \mapsto \overline{v'}, \overline{z} \mapsto \overline{0}] \cdot \ell''') \mid C, 0, C'' : \mathsf{S}_\mathtt{m}\end{array}}\ [\textsc{meth}]$$

$$\frac{\begin{array}{c}[\![e]\!]_{C',v,C,\ell} = C'' \quad [\![\overline{e'}]\!]_{C',v,C,\ell} = \overline{v'} \\ \mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ z;\mathsf{S}_\mathtt{m}}\} \notin \mathit{typeof}(C'') \quad \mathsf{fallback} \in \mathit{typeof}(C'')\end{array}}{C(\ell \cdot \ell') \mid C', v, C : e.\mathtt{m}(\overline{e'}) \xrightarrow{\checkmark} C(\ell \cdot \ell) \mid 0}\ [\textsc{meth-fb}]$$

$$\frac{\begin{array}{c}[\![e]\!]_{C',v,C,\ell} = C'' \quad [\![\overline{e'}]\!]_{C',v,C,\ell} = \overline{v'} \\ \mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ z;\mathsf{S}_\mathtt{m}}\}, \mathsf{fallback} \notin \mathit{typeof}(C'')\end{array}}{C(\ell \cdot \ell') \mid C', v, C : e.\mathtt{m}(\overline{e'}) \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid 0}\ [\textsc{meth-err}]$$

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

$$\frac{\begin{array}{c}[\![e]\!]_{C',v,C,\ell} = C'' \quad [\![\overline{e'}]\!]_{C',v,C,\ell} = \overline{v'} \quad [\![e'']\!]_{C',v,C,\ell} = v'' \\ \mathtt{m}(\overline{\mathsf{T}\ x})\ \mathsf{payable}\ \{\overline{\mathsf{T}'\ z;\mathsf{S}_\mathtt{m}}\} \notin \mathit{typeof}(a) \quad \mathsf{fallback} \in \mathit{typeof}(C'') \\ \ell(\mathsf{balance}) \geqslant v'' \quad \ell_1 = \ell[\mathsf{balance} \mapsto^- v''] \quad \ell_2 = \ell''[\mathsf{balance} \mapsto^+ v'']\end{array}}{C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C', v, C : e.\mathtt{m}.\mathsf{value}(e'')(\overline{e'}) \xrightarrow{\checkmark} C(\ell_1 \cdot \ell_1) \mid C''(\ell_2 \cdot \ell_2) \mid 0}\ [\textsc{meth-pay-fb}]$$

$$\frac{\begin{array}{c}[\![e]\!]_{C',v,C,\ell} = C'' \quad [\![\overline{e'}]\!]_{C',v,C,\ell} = \overline{v'} \quad [\![e'']\!]_{C',v,C,\ell} = v'' \\ \ell(\mathsf{balance}) < v'' \quad \text{or} \quad \Big(\mathtt{m}(\overline{\mathsf{T}\ x})\ \mathsf{payable}\ \{\overline{\mathsf{T}'\ z;\mathsf{S}_\mathtt{m}}\}, \mathsf{fallback} \notin \mathit{typeof}(C'')\Big)\end{array}}{C(\ell \cdot \ell') \mid C''(\ell'' \cdot \ell''') \mid C', v, C : e.\mathtt{m}.\mathsf{value}(e'')(\overline{e'}) \xrightarrow{\mathsf{fail}} C(\ell' \cdot \ell') \mid C''(\ell''' \cdot \ell''') \mid 0}\ [\textsc{meth-pay-err}]$$

Fig. 5. Semantics of μSolidity.

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 method invocations, namely [METH-PAY], [METH-PAY-FB] and [METH-PAY-ERR] account for invocations of payable methods. In these case 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, namely either when caller's balance is smaller than the value to be sent or when the dispatch cannot be done because there is no method and there is no fallback.

The semantics of other statements is standard and therefore the comments are omitted. We conclude by noticing that blockchain-statements are executed *sequentially* and in a *deterministic way* (because they originated in smart contracts' methods).

Initial states. The initial state of a μSolidity program $\left(\mathsf{C}_1, \cdots, \mathsf{C}_n\right)$ with environment Γ, caller C, callee D, transmitted value v and runtime statement S is (the notation keeps the program implicit):

$$\mathit{state}_\Gamma(C, v, D, \mathsf{S}) \stackrel{\text{def}}{=} \textstyle\prod_{i\in 1..n} C_i(\ell_i \cdot \ell_i) \mid C, v, D : \mathsf{S}$$

where $\ell_i \stackrel{\text{def}}{=} \Gamma(C_i)$ and $\mathit{typeof}(C_i) = \mathsf{C}_i$ for $1 \leqslant i \leqslant n$. Given a state $\mathcal{S} = \prod_{i\in 1..n} C_i(\ell'_i \cdot \ell_i) \mid C_k, v, C_h : \mathsf{S}$, we define

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

Proposition 1. *For every $\mathcal{S} = \prod_{i\in 1..n} C_i(\ell'_i \cdot \ell_i) \mid C, v, D : \mathsf{S}$ there is Θ such that $\mathit{envs}(\mathcal{S}) \vdash^{v}_{C,D} \mathsf{S} : \Theta$. This judgment will be abbreviated $\mathcal{S} \vdash \Theta$ below.*

Lemma 1 (Substitution Lemma). *Let Γ_0, Γ_1 be pure environments and Γ, Γ' be ground environments. If $\Gamma_0, \Gamma_1[\overline{x} \mapsto \overline{x_0}, \overline{y} \mapsto \overline{0}] \vdash^{z}_{C,D} \mathsf{S} : \Theta$ then $\Gamma, \Gamma'[\overline{x} \mapsto \overline{v}, \overline{y} \mapsto \overline{0}] \vdash^{u}_{C,D} \mathsf{S} : \Theta\{^{u,\overline{v}}/_{z,\overline{x_0}}\}[\Gamma_0, \Gamma_1 \rightsquigarrow \Gamma, \Gamma']$. Similarly for expressions.*

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

The next Theorem requires a small extension of the translation in Figure 3 to cover the case of states with runtime statements $\mathsf{0}$:

$$\frac{}{\Gamma, \Gamma \vdash^{v}_{C,D} \mathsf{0} : \Gamma}\ [\text{ZERO}]$$

(v, C, and D are generic). Also in these cases, we write $\mathcal{S} \vdash \Theta$.

Theorem 1 *Let $\left(\mathsf{C}_1, \cdots, \mathsf{C}_n\right)$ be a μSolidity program and $\vdash \left(\mathsf{C}_1, \cdots, \mathsf{C}_n\right) : \mathcal{D}$. Let also $\mathcal{S}$ be an initial state and $\mathcal{S} \vdash \Theta$. Then*

1. *(*determinism*) If $\Theta \Longrightarrow^{*}_{\mathcal{D}} \Theta'$ then there is at most one Θ'' such that $\Theta' \Longrightarrow_{\mathcal{D}} \Theta''$;*

2. *(*correctness*) If* $\mathcal{S} \longrightarrow^* \mathcal{S}'$ *and* $\mathcal{S}' \vdash \Theta'$ *then* $\Theta \Longrightarrow^*_{\mathcal{D}} \Theta''$*, where* Θ'' *is either equal to* Θ' *or* $\Theta'' = (C_h = C_h)\Theta' + \sum_{i\in(1..n)\setminus h}(C_h = C_i)\Theta_i$*. Similarly for* $\xrightarrow{\checkmark}$ *and* $\xrightarrow{\mathsf{fail}}$*.*

Proof. The proof is by induction on the length of $state_\Gamma(C, v, D, \mathsf{S}) \longrightarrow^* \mathcal{S}'$. We use the property that, if $\Gamma \vdash^v_{C,D} e' : e''$ and $[\![e]\!]_{C,v,D,\Gamma(D)} = v'$ then e'' is a constant expression (without any variable) that, if computed, gives v'.

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

$$state_\Gamma(C, v, D, \mathsf{S}) \longrightarrow^* \mathcal{S} \longrightarrow \mathcal{S}'$$

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

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

(1) $\mathcal{S} = \prod_{i\in 1..n} C_i(\ell'_i \cdot \ell_i) \mid C_j, v, C_k : e.\mathtt{m}(\overline{e'})$
(2) $[\![e]\!]_{C_j,v,C_k,\ell'_k} = C_h$
(3) $[\![\overline{e'}]\!]_{C_j,v,C_k,\ell'_k} = \overline{v'}$
(4) $\mathtt{m}(\overline{\mathsf{T}\ x})\{\overline{\mathsf{T}'\ z;}\mathsf{S}_\mathtt{m}\} \in typeof(\mathsf{C}_h)$
(5) $\Gamma, \Gamma' = envs(\mathcal{S})$

Additionally, since $\mathcal{S} \vdash \Theta$, we have used rule [INVK-NV] with the hypotheses:

(6) $\Gamma' \vdash^v_{C_j,C_k} e : e_0$
(7) $\Gamma' \vdash^v_{C_j,C_k} \overline{e'} : \overline{e''}$

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

According to rule [METH], we have

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

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

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

and, by the Substitution Lemma, we obtain

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

By definition (8) is exactly $\mathcal{S}' \vdash \Theta_{C_k}\{^{\overline{v}}/_{\overline{x_0}}\}[\Gamma_0, \Gamma_1 \rightsquigarrow \Gamma, \Gamma']$.

As regards Θ, we observe that

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

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

$$\Theta \Longrightarrow_{\mathcal{D}} \Theta_{C_k}\{^{\overline{v}}/_{\overline{z}}\}[\Gamma_0, \Gamma_1 \rightsquigarrow \Gamma, \Gamma']$$

(in this case there is no variable representing the transferred Ether value) where $\Theta' = \sum_{C \in Id}(H = C)\ \Theta_C$. This concludes the proof.

B The extension with addresses and continuations

Booleans and Addresses. We extend types T with addresses and maps as follows:

$$\mathsf{T} ::= \quad \mathsf{uint} \quad | \quad \mathsf{bool} \quad | \quad \mathsf{address}$$

where address is the set of *contract addresses*. The extension of the encoding in Section 4 to cope with these new features is not difficult, given that the set of addresses is finite. Below we report the rule when types are either uint or bool (ranged over by T in the following) or address .

[FUNCTION-ADDR]

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

where $\overline{k}$ is a sequence of elements that are either 0 or *true*, according to the type in $\overline{\mathsf{T}'}$ is uint or bool .

Continuations. Extending the language in Section 2 with continuations is not easy, in particular because we need to output cost equations whose functions have no explicit continuation. Below we report the current solution of our analyzer that only admits continuations with non-recursive functions.

We extend the μSolidity syntax as follows:

$$\mathsf{C} ::= \quad \mathsf{contract}\ C\ \{\quad \overline{\mathsf{T}\ \mathtt{f};}\quad \mathsf{F}\quad \left[\mathsf{fallback}(\ )\ \mathsf{payable}\ \{\overline{\mathsf{T}\ x;}\ \mathsf{S}\}\right]\ \}$$

$$\mathsf{F} ::= \quad \cdots \quad | \quad \mathsf{function}\ \mathtt{m}(\overline{\mathsf{T}x})[\mathsf{payable}]\ \mathsf{returns}\ (\mathsf{T})\ \{\overline{\mathsf{T}\ y;}\ \mathsf{S}\}\quad \mathsf{F}$$

$$\begin{array}{rl}\mathsf{S} ::= & \cdots \quad | \quad \mathsf{if}\ (\mathsf{E})\ \{\ \mathsf{S}\ \}\ \mathsf{else}\ \{\ \mathsf{S}\ \};\ \mathsf{S} \quad | \quad \mathsf{return}(\mathsf{E}) \quad | \quad \mathsf{return}(\mathsf{E}.\mathtt{m}[.\mathsf{value}(\mathsf{E})](\overline{\mathsf{E}})) \\ & | \quad \mathsf{E}.\mathtt{m}[.\mathsf{value}(\mathsf{E})](\overline{\mathsf{E}});\ \mathsf{S} \quad | \quad x = \mathsf{E}.\mathtt{m}[.\mathsf{value}(\mathsf{E})](\overline{\mathsf{E}});\ \mathsf{S}\end{array}$$

In particular, the fallback functions in contracts may now have nonempty bodies; function bodies may also return values; function invocations, as well as conditionals, may have continuations.

There is a restriction that is relevant for our analysis: *we only admit tail recursive (or tail mutual recursive) invocations.* This means that, if we take

non-recursive functions in a program, then the sequence of possible nested invocations is finite and its upper bound is statically determined. Technically, we compute the graph of invocations and verify that it is acyclic, i.e. we give weight 0 to tail invocations and 1 to the other invocations; the graph must have paths of finite weight, otherwise the program is rejected. In the same line, since in Solidity fallback functions are invoked in correspondence of Ether transfer and failures of function dispatch, we drop function invocations from fallback bodies. In the following we assume that μSolidity programs comply with the foregoing constraints.

In order to encode nested invocations of a μSolidity program $(\mathsf{C}_1, \cdots, \mathsf{C}_n)$ we use *stacks*. Stacks σ have syntax

$$\sigma ::= \quad \langle \bot, \mathtt{m}_\bot, \overline{\bot}, \bot, 0 \rangle \quad | \quad \langle C, \mathtt{m}, \overline{e}, D, e' \rangle :: \sigma \ .$$

where $\bot$, $\mathtt{m}_\bot$ are special names and we assume that expressions e also contain $\bot$. Every element of the stack is a tuple *of the same length*, which is $\iota = 4$, where ι is the maximal number of arguments and local variables of a function in $(\mathsf{C}_1, \cdots, \mathsf{C}_n)$. It is worth to notice that stacks have always a final tuple such that the first element is $\bot$. We use the following operations on tuples of expressions:

- $|\overline{e}|$ returns the length of the tuple;
- $\overline{e} \downarrow_{D.\mathtt{m}}$ returns the prefix of $\overline{e}$ whose length is equal to the number of arguments of $D.\mathtt{m}$.

Figures 6 and 7 report the rules for translating μSolidity programs in intermediate codes. Rule [EMPTY-CONT] deals with empty statements. It has two subcases, according to the stack is empty or not. According to our model, the stack is empty when there is no other continuation to execute, therefore the first element of the tuple (e.g. H) is $\bot$ and we return the current environment. In the second subcase, we evaluate the continuation, say $D.\mathtt{m}$ (in the judgment in the premise we drop out useless expressions in the stack). That is, in our translation, continuations of invocations and conditionals are managed by ad-hoc functions that extends those of the μSolidity program. For this reason, in the premise of [EMPTY-CONT] the continuation is evaluated in a current environment where balances of the caller and the callee are respectively increased and decreased by the value e'. This is done because the judgment in the premise is an invocation, which will perform the opposite operations, see rule [INVK-TAIL]). Rule [EMPTY-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 stack misses the first argument that is provided by the return and is therefore taken by $\mathtt{m}_\mathsf{S}$. Rules [ASGN-CONT-INVK] and [INVK-CONT] define invocations with continuations when invocations return a value or when the function is void. In both cases, we assume the functions of the corresponding smart contract are extended with new ad-hoc functions managing the continuation. The formal parameters of these ad-hoc functions are the variables in the current environments (e.g. $dom(\Gamma'_{Var})$, see also rule [FUNCTION-CONT]) plus an additional variable for function returning a value. The stack store the values of

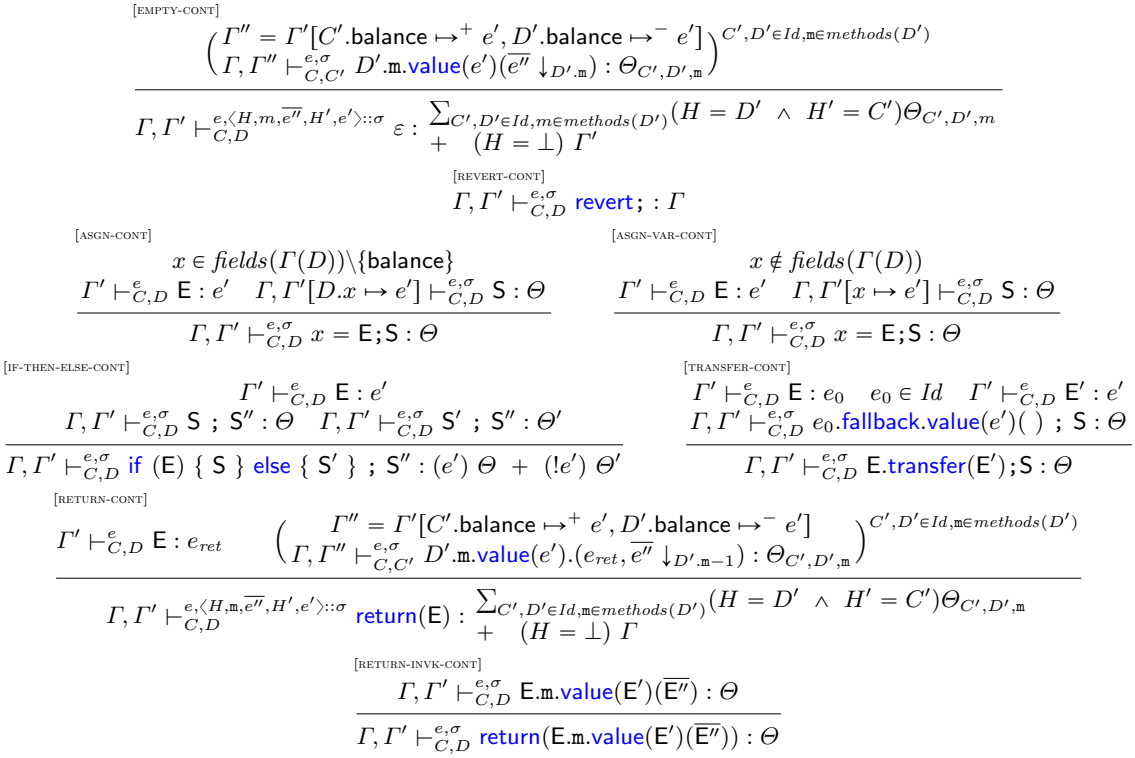

Fig. 6. Translation of μSolidity statements with continuations, Part I

the variables in the environment, which will be restored when the continuation is triggered (see rules [EMPTY-CONT] and [RETURN-CONT]).

Rules [INVK-TAIL-NV] and [INVK-TAIL] extend rules [INVK-NV] and [INVK] of Figure 3 with the management of stacks.

B.1 The analysis

Similarly to what we have done in Section 4, we need to make the intermediate code adequate to `CoFloCo`. We stick to the same cost model defined before and below we only detail the differences with Section 4.

Let $(\mathsf{C}_1, \cdots, \mathsf{C}_n)$ be such a program and let κ be the maximal weight of a path in the graph of invocations. Let also ι be the maximal number of arguments and local variables of a function in $(\mathsf{C}_1, \cdots, \mathsf{C}_n)$. We let $\overline{\sigma}$ be a tuple whose

[ASGN-CONT-INVK]

$$\frac{\begin{array}{c}\Gamma' \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in Id \quad [\Gamma' \vdash^{e}_{C,D} \mathsf{E}' : e'] \quad \Gamma' \vdash^{e}_{C,D} \overline{\mathsf{E}''} : \overline{e''} \\ \mathsf{function}\ \mathtt{m}(\overline{\mathsf{T}_y\ y})\ [\mathsf{payable}]\ \mathsf{returns}\ (\mathsf{T}')\ \{\overline{\mathsf{T}_z\ z};\ \mathsf{S}_m\} \in e_0 \\ dom(\Gamma'|_{Var}) = \overline{w} \quad \mathsf{function}\ \mathtt{m}_{\mathsf{S}}(\mathsf{T}'\ x', \overline{\mathsf{T}_w\ w})\ \mathsf{payable}\ \{x = x';\mathsf{S}\} \in D \\ \Gamma'(\overline{w}) = \overline{e_w} \qquad |\overline{\bot}| = \iota - |\overline{e_w}| \qquad \sigma' = \langle D, m_{\mathsf{S}}, \overline{e_w}, \overline{\bot}, C, e\rangle :: \sigma \\ \Gamma, \Gamma' \vdash^{e,\sigma'}_{C,D} e_0.\mathtt{m}[.\mathsf{value}(e')](\overline{e''}) : \Theta\end{array}}{\Gamma, \Gamma' \vdash^{e,\sigma}_{C,D} x = \mathsf{E}.\mathtt{m}[.\mathsf{value}(\mathsf{E}')](\overline{\mathsf{E}''});\mathsf{S} : \Theta}$$

[INVK-CONT]

$$\frac{\begin{array}{c}\Gamma' \vdash^{e}_{C,D} \mathsf{E} : e_0 \quad e_0 \in Id \quad [\Gamma' \vdash^{e}_{C,D} \mathsf{E}' : e'] \quad \Gamma' \vdash^{e}_{C,D} \overline{\mathsf{E}''} : \overline{e''} \\ \mathsf{function}\ \mathtt{m}(\overline{\mathsf{T}_y\ y})\ [\mathsf{payable}]\ \{\overline{\mathsf{T}_z\ z};\ \mathsf{S}_m\} \in e_0 \\ dom(\Gamma'|_{Var}) = \overline{w} \quad \mathsf{function}\ \mathtt{m}_{\mathsf{S}}(\overline{\mathsf{T}_w\ w})\ \mathsf{payable}\ \{\mathsf{S}\} \in D \\ \Gamma'(\overline{w}) = \overline{e_w} \qquad |\overline{\bot}| = \iota - |\overline{e_w}| \qquad \sigma' = \langle D, m_{\mathsf{S}}, \overline{e_w}, \overline{\bot}, C, e\rangle :: \sigma \\ \Gamma, \Gamma' \vdash^{e,\sigma'}_{C,D} e_0.\mathtt{m}[.\mathsf{value}(e')](\overline{e''}) : \Theta\end{array}}{\Gamma, \Gamma' \vdash^{e,\sigma}_{C,D} \mathsf{E}.\mathtt{m}[.\mathsf{value}(\mathsf{E}')](\overline{\mathsf{E}''});\mathsf{S} : \Theta}$$

[INVK-TAIL-NV]

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

[INVK-TAIL]

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

[FUNCTION-CONT]

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

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

length is $(\iota + 4) \times (\kappa + 1)$ that is defined as follows:

$$\widehat{\epsilon} = \underbrace{\bot, \cdots, \bot}_{(\iota+4)\times(\kappa+1)\ \text{times}}$$

$$\widehat{\alpha_1 :: \cdots :: \alpha_h} = \widehat{\alpha_1}, \overset{25}{\cdots}, \widehat{\alpha_h}, \underbrace{\bot, \cdots, \bot}_{(\iota+4)\times(\kappa+1-h)\ \text{times}}$$

where $\overbrace{\langle C, m, \overline{e}, D, e' \rangle} = C, m, \overline{e}, \underbrace{\bot, \cdots, \bot}_{\iota - |\overline{e}| \text{ times}}, D, e'$. Sequences of $(\iota + 4) \times (\kappa + 1)$ elements will be ranged over by φ.

Cost equations of a μSolidity program are derived by the corresponding intermediate code as follows:

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

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

2. Let Θ be the main type and let $\bigvee_{i \in 1..k} (\varphi'_i)\ \Theta'_i$ be the canonical form of $\ulcorner \Theta \urcorner$. Then the cost equations of a program that starts with environments Γ and Γ' also include

$$\begin{array}{lcll} \mathit{main}(\lfloor \Gamma \rfloor, \lfloor \Gamma' \rfloor, 0, \overbrace{\varepsilon}) = & \Theta'_1 & [\varphi'_1] \\ & \cdots & \\ \mathit{main}(\lfloor \Gamma \rfloor, \lfloor \Gamma' \rfloor, 0, \overbrace{\varepsilon}) = & \Theta'_k & [\varphi'_k] \quad . \end{array}$$