RESEARCH-ARTICLE

A Sound Type System for Secure Currency Flow

LUCA ACETO, Reykjavík University, Reykjavik, Iceland

DANIELE GORLA, Sapienza University of Rome, Rome, RM, Italy

STIAN LYBECH, Reykjavík University, Reykjavik, Iceland

Open Access Support provided by:

Sapienza University of Rome

Reykjavík University

Published: 21 March 2026
Online AM: 29 December 2025
Accepted: 16 December 2025
Revised: 07 August 2025
Received: 05 March 2025

A Sound Type System for Secure Currency Flow

LUCA ACETO, Reykjavík University, Reykjavík, Iceland and Gran Sasso Science Institute, L'Aquila, Italy
DANIELE GORLA, Dip. Informatica, Sapienza Univ. di Roma, Roma, Italy
STIAN LYBECH, Reykjavík University, Reykjavík, Iceland

In this article, we focus on TinySol, a minimal calculus for Solidity smart contracts, introduced by Bartoletti, Galletta and Murgia. We start by rephrasing its syntax (to emphasise its object-oriented flavour) and give a new big-step operational semantics for that language. We then use it to define two security properties, namely call integrity and noninterference. These two properties have some similarities in their definition, in that they both require that some part of a program is not influenced by the other part. However, we show that the two properties are actually incomparable. Nevertheless, we provide a type system that statically ensures both noninterference and call integrity; hence, well-typed programs satisfy both properties. We finally discuss the practical usability of the type system and its limitations by means of some simple examples.

CCS Concepts: • **Theory of computation → Type theory**; • **Security and privacy → Information flow control**;

Additional Key Words and Phrases: smart contracts, blockchain, call integrity, noninterference, type system, program analysis

ACM Reference format:
Luca Aceto, Daniele Gorla, and Stian Lybech. 2026. A Sound Type System for Secure Currency Flow. *ACM Trans. Program. Lang. Syst.* 48, 1, Article 4 (March 2026), 56 pages.
https://doi.org/10.1145/3786766

1 Introduction

The classic notion of noninterference [23] is a well-known concept that has been applied in a variety of settings to characterise both integrity and secrecy in programming. In particular, this property has been defined in [53] by Volpano et al. in terms of a lattice model of security levels (e.g., 'High' and 'Low', or 'Trusted' and 'Untrusted'); the key point is that information must *not* flow from a higher to a lower security level. Thus, the lower levels are unaffected by the higher ones, and, conversely, the higher levels are 'noninterfering' with the lower ones.

Ensuring noninterference seems particularly relevant in a setting where not only information but also *currency* flows between programs. This is a core feature of *smart contracts*, which are programs that run on top of a blockchain and are used to manage financial assets of users, codify transactions, and implement custom tokens; see e.g., [45] for an overview of the architecture. The code of a smart contract resides on the blockchain itself and is therefore both immutable and publicly visible. This

The work of L. Aceto and S. Lybech was partially supported by the Icelandic Research Fund Grant No. 218202-05(1-3). S. Lybech also received financial support from the Department of Computer Science at Reykjavik University for the period January–May 2025.
Authors' Contact Information: Luca Aceto, Reykjavík University, Reykjavík, Iceland and Gran Sasso Science Institute, L'Aquila, Italy; e-mail: luca@ru.is; Daniele Gorla (corresponding author), Dip. Informatica, Sapienza Univ. di Roma, Roma, Italy; e-mail: gorla@di.uniroma1.it; Stian Lybech, Reykjavík University, Reykjavík, Iceland; e-mail: stian21@ru.is.

ACM 1558-4593/2026/3-ART4
https://doi.org/10.1145/3786766

```
contract X {                                contract Y {
  ⋮                                           ⋮
  field called := F;                          deposit(x) {
  transfer(y) {                                 x.transfer(this)$0
    if ¬this.called ∧                         }
       this.balance ≥ 1                     }
      then y.deposit(this)$1;
           this.called := T
    else skip
  }
}
```

Fig. 1. Illustration of reentrancy written in the language TinySol.

is one of the important ways in which the 'smart-contract programming paradigm' differs from conventional programming languages.

Public visibility means that *vulnerabilities* in the code can be found and exploited by a malicious user. Moreover, if a vulnerability is discovered, immutability prevents the contract creator from correcting the error. Thus, it is obviously desirable to ensure that a smart contract is safe and correct *before* it is deployed on the blockchain.

The combination of immutability and visibility has led to huge financial losses in the past [see, e.g., 6, 17, 50]. A particularly spectacular example was the infamous DAO attack on the Ethereum platform in 2016, which led to a loss of 60 million dollars [17]. This was made possible because a certain contract (the DAO contract, storing assets of users) was *reentrant*, that is, it allowed itself to be called back by the recipient of a transfer *before* recording that the transfer had been completed.

Reentrancy is a pattern based on mutual recursion, where one method f calls another method g while also transferring an amount of currency along with the call. If g then immediately calls f back, it may yield a recursion where f continues to transfer funds to g. We can illustrate the problem as in Figure 1, using a simple, imperative and class-based model language called TinySol [7]. This model language, which we shall formally describe in Section 2, captures some of the core features of the smart-contract language Solidity [20], which is the standard high-level language used to write smart contracts for the Ethereum platform. This language resembles an object-oriented language, where contracts are autonomous entities that are characterised by their fields (representing the local state) and methods (representing the functionalities offered); however, no notion of class or of inheritance is present. A key feature of this language is that contracts have an associated `balance`, i.e., a specific field representing the amount of currency stored in each contract, which cannot be modified *except* through method calls to other contracts. Each method call has an extra parameter, representing the amount of currency to be transferred along with the call, and a method call thus represents a (potential) outgoing currency flow.

In Figure 1, `X.transfer(y)` first performs a sanity check to ensure that it has not already been called and that the contract contains sufficient funds, which are stored in the field `balance`. Then it calls `y.deposit(this)` and transfers `1` unit of currency along with the call, where `y` is the address received as parameter. However, suppose that the received address is `Y`. Then `Y.deposit(x)` immediately calls `X.transfer(y)` back, with `this` as the actual parameter, thus yielding a mutual recursion, because the field `called` will never be set to `T`. A transaction that invokes `X.transfer(Y)` with any number of currency units will trigger the recursion.

The problem is that currency cannot be transferred without also transferring control to the recipient, and execution of `X.transfer(y)` comes to depend on unknown and untrusted code in the contract residing at the address received as the actual parameter. Simply swapping the order of lines 6 and 7 in `X` solves the problem in this particular case, but it might not always be possible to move external calls to the last position in a sequence of statements. Furthermore, the execution of a function f can also depend on external fields and not only on external calls. Thus, reentrancy is not just a purely syntactic property.

The property of reentrancy in Ethereum smart contracts has been formally characterised by Grishchenko et al. in [26]. Specifically, they define another property, named *call integrity*, which implies the absence of reentrancy (see [26, Theorem 1]) and has been identified in the literature as one of the safety properties that smart contracts should have. Informally, this property requires any call to a method in a 'trusted' contract (say, X) to yield the very same sequence of currency flows (i.e., method calls) even if some of the other 'untrusted' contracts (or their stored values) are changed. In a sense, the code and values of the other contracts, which could be controlled by an attacker, must not be able to affect the currency flow from X.

A disadvantage of the definition of call integrity given in [26] is that it relies on a universal quantification over all possible execution contexts, making it difficult to check in practice. However, call integrity seems intuitively to be related to noninterference, in the sense that both stipulate that changes in one part of a program should not have an effect upon another part. Even though we discover that the two properties are incomparable, one might hope to be able to apply techniques for ensuring noninterference to also capture call integrity. Specifically, in [53], Volpano et al. show that noninterference can be soundly approximated using a type system. In the present article, we shall therefore create an adaptation of that type system for secure-flow analysis to the setting of smart contracts and show that the resulting type system *also* captures call integrity.

To recap, our main contributions in this article are: (1) a thorough study of the connections between call integrity and noninterference for smart contracts written in the language TinySol, and (2) a sound proof-technique for both properties, in the form of a type system guaranteeing both noninterference and call integrity for programs written in that language. We choose TinySol because it provides a minimal calculus for Solidity contracts and thus allows us to focus on the gist of our main contributions in a simple setting. In doing this, we also provide a more standard operational semantics for this language than the one given in [7]; this can be considered a third contribution of our work.

The article is organised as follows: In Section 2, we describe a revised version of the smart-contract language TinySol [7]. In Section 3, we adapt the definition of call integrity from [26] and of noninterference from [47] to this language; we then show that these two desirable properties are actually incomparable. Nevertheless, there is an overlap between them. In Section 4, we create a type system for ensuring noninterference in TinySol, along the lines of Volpano et al. [53], and prove the corresponding type soundness results (Theorems 3–5). Our main results are Corollary 1 and Theorem 6, which show that well-typedness provides a sound approximation to *both* noninterference *and* call integrity. This is used on a few examples in Section 5, where we also discuss the limitations of the type system. We survey some related work in Section 6 and conclude the article with some directions for future research in Section 7. Some of the proofs are relegated to the appendix, to streamline reading; moreover, Appendix G contains a synopsis of all the symbols and notations used in this article.

With respect to the conference version [1], this article extends our syntax of TinySol with the construct of *delegate call*, which allows us to write more sophisticated programming examples (see, e.g., the *pointer to implementation pattern* provided in Figure 3). Moreover, here we introduce a static inheritance between contract types; this is done through a different (w.r.t. the conference

$$
\begin{array}{rcl}
DF \in \mathsf{Dec}_F & ::= & \epsilon \mid \texttt{field}\ p\ \texttt{:=}\ v\texttt{;}DF \\
DM \in \mathsf{Dec}_M & ::= & \epsilon \mid f(\widetilde{x})\ \texttt{\{}\ S\ \texttt{\}}\ DM \\
DC \in \mathsf{Dec}_C & ::= & \epsilon \mid \texttt{contract}\ X\ \texttt{\{} \\
 & & \qquad \texttt{field balance := } z\texttt{;}\ DF \\
 & & \qquad \texttt{send() \{ skip \}}\ DM \\
 & & \texttt{\}}\ DC \\
m \in \mathsf{MVar} & ::= & \texttt{this} \mid \texttt{sender} \mid \texttt{value} \\
L \in \mathsf{LVal} & ::= & x \mid \texttt{this.}p \\
e \in \mathsf{Exp} & ::= & v \mid x \mid m \mid e\texttt{.balance} \mid e\texttt{.}p \mid \mathrm{op}(\widetilde{e}) \\
S \in \mathsf{Stm} & ::= & \texttt{skip} \mid \texttt{throw} \mid \texttt{var}\ x\ \texttt{:=}\ e\ \texttt{in}\ S \mid L\ \texttt{:=}\ e \mid S_1\texttt{;}S_2 \mid \texttt{if}\ e\ \texttt{then}\ S_{\mathsf{T}}\ \texttt{else}\ S_{\mathsf{F}} \\
 & & \mid \texttt{while}\ e\ \texttt{do}\ S \mid e_1\texttt{.}f(\widetilde{e})\texttt{\$}e_2 \mid \texttt{dcall}\ e\texttt{.}f(\widetilde{e}) \\
v \in \mathsf{Val} & ::= & \mathbb{Z} \cup \mathbb{B} \cup \mathsf{ANames}
\end{array}
$$

where $x, y \in \mathsf{VNames}$ (variable names), $p \in \mathsf{FNames}$ (field names),
$A, X, Y, Z \in \mathsf{ANames}$ (address names), $f, g \in \mathsf{MNames}$ (method names)

Fig. 2. The syntax of TinySol.

version) notion of *interfaces* (see Definition 8) and yields a different, and in a sense simpler, notion of subtyping (see Figures 9 and 10). Also, the type system itself, albeit equivalent to the one of the conference version, is presented here in a more elegant and readable way. Finally, this article offers proofs of all the results announced in [1] and we also add and prove here a standard subject reduction result.

2 The TinySol language

In [7], Bartoletti et al. present the TinySol language, a standard imperative language (similar to the classic `While` language [37]), extended with classes (contracts) and two constructs: (1) a `throw` command, representing a fatal error and (2) a procedure call with an extra parameter z, denoting an amount of some digital asset that is transferred along with the call from the caller to the callee. TinySol captures (some of) the core features of Solidity, and, in particular, it is sufficient to represent the essential pattern of reentrancy.

In this section, we present a version of TinySol that has been adapted to facilitate our later developments of the type system. Compared to the presentation in [7], we have, in particular, added explicit declarations of variables (local to the scope of a method) and fields (corresponding to the *keys* in the original presentation) to have a place for type annotations in the syntax. Furthermore, differently from [1, 7], here we also consider a second kind of procedure call taken from Solidity (named *delegate call*), which allows a method to be executed in the environment of the caller; i.e., *as if* the method was declared in the caller contract, rather than in the callee contract.

2.1 Syntax

The syntax of TinySol is given in Figure 2, where we use the notation $\widetilde{\cdot}$ to denote (possibly empty) sequences of items. The set of *values*, ranged over by v, consists of the sets of integers $\mathbb{Z}$ ranged over by z, booleans $\mathbb{B} = \{\mathsf{T}, \mathsf{F}\}$ ranged over by b and address names ANames ranged over by X, Y.

We introduce explicit declarations for fields DF, methods DM, and contracts DC. The latter also includes account declarations: an *account* is a contract that contains only the declarations of a special field `balance` and of a single special method `send()`, which does nothing and is used only

to transfer funds to the account. In contrast, a contract usually contains other declarations of fields and methods. For simplicity, we make no syntactic distinction between an account and a contract, but, for the purpose of distinguishing, we can assume that the set ANames is divided into contract addresses and account addresses.

We have four 'magic' keywords in our syntax:

- —`balance`, a special integer field that records the current balance of the contract (or account). It can be read from, but not directly assigned to, except through method calls. This ensures that the total amount of currency in the blockchain remains constant during execution.
- —`value`, a special integer variable that is bound to the currency amount transferred with a method call.
- —`sender`, a special address variable that is always bound to the address of the caller of a method.
- —`this`, a special address variable that is always bound to the address of the contract that contains the method that is currently executed.

The last three of these are local variables, and we collectively refer to them as 'magic variables' $m \in \mathsf{MVar}$. The declarations of variables and fields are very similar: the main difference is that variable bindings will be created at runtime (and with scoped visibility). Hence, we can let the initial assignment of a local variable be an *expression* e; in contrast, the initial assignment to a field must be a *value* v (see the grammar for Dec_F in Figure 2).

The core of the language is the declaration of the expressions e and of the statements S, which are almost the same as in [7]: expressions are made up by values, variables (possibly including the magic ones), field accesses and operations op (that we leave unspecified here); statements include a no-op instruction, one for raising exceptions (we do not consider exception handling in this article), local variable declarations and the usual instructions for variable assignments, concatenating statements, conditional choice, loop and two flavours of method calls. The main differences with respect to [7] are: (1) we introduce fields p in expressions, instead of keys; (2) we explicitly distinguish between (global) fields and (local) variables, where the latter are declared with a scope limited to a statement S and (3) we introduce explicit *lvalues* L, to restrict what can appear on the left-hand side of an assignment (in particular, this ensures that the special field `balance` can never be assigned directly).

Besides the ordinary method call, Solidity also features another kind of method call with a different calling style, known as the *delegate call*. The purpose of this construct is to facilitate code reuse and the creation of contract libraries. This is desirable, since the cost of deploying a smart contract on the Ethereum chain depends on the code size[1], which therefore encourages developers to use libraries for common tasks. The special feature of this call style is that the method body is executed in the context of the *caller* contract, rather than in the context of the contract in which it is declared, which allows the calling contract to use the called method *as if* it were part of the calling contract itself. A natural way to use delegate calls would be as a body of a stub method declaration like:

$$\texttt{f(}\tilde{x}\texttt{) \{dcall X.f(}\tilde{x}\texttt{)\}},$$

which simply forwards the call to an implementation that resides in `X`, rather than locally.

A related usage is the so-called *pointer to implementation* pattern described in [18], which is illustrated in Figure 3. This pattern can be used to allow a contract to update its implementation, despite the immutable nature of a blockchain. It consists of a contract `Y`, which acts as a proxy for the functionality provided by a different contract that resides at the address `X`. This address

[1]See the description of the `CREATE` and `CREATE2` opcodes in [48].

```
contract Y {
  field owner := A;
  field impl := X;
  ...
  Update(addr) {if sender = this.owner then this.impl := addr else skip}
  f_1(x̃) { dcall impl.f_1(x̃) }
  ⋮
  f_h(x̃) { dcall impl.f_h(x̃) }
}
```

Fig. 3. The *pointer to implementation* pattern.

is stored in the field `impl`, and `Y` simply forwards all calls to methods $f_1, \ldots f_h$ to their actual implementations in `X`. Finally, `Y` provides an `Update` method, which allows the contract owner `A` to overwrite the address stored in `impl`. Thus, if the actual implementation needs to be updated, `A` can deploy a new contract `X'` and, by invoking `Update(X')`, he can update the pointer to the implementation in `Y` without affecting any user of the contract.

Finally, as in the original presentation of TinySol, we can also use our new formulation of the language to describe transactions and blockchains. A *transaction* is simply a sequence of calls, where the caller is an account A, rather than a contract. We denote this by writing $A \overset{z}{\rightsquigarrow} X.f(\tilde{v})$, which expresses that the account A calls the method f on the contract (residing at address) X, with actual parameters $\tilde{v}$, and transferring z units of currency with the call. We can then model blockchains as follows.

Definition 1 *(Syntax of Blockchains).* A *blockchain* $BC \in \mathcal{BC}$ is a list of initial contract declarations DC, followed by a transaction $\tau \in \mathsf{Tr}$:

$$BC ::= DC\ \ \tau \qquad\qquad \tau ::= \epsilon \mid A \overset{z}{\rightsquigarrow} X.f(\tilde{v}), \tau$$

Notationally, a blockchain with an empty DC will be simply written as the sequence of transactions. ■

2.2 Big-Step Semantics

To define the semantics of TinySol, we need some environments to record the bindings of variables (including the three magic variable names `this`, `sender` and `value`), fields, methods, and contracts. We define them as sets of partial functions as follows.

Definition 2 *(Binding Model).* We define the following sets of partial functions:

$$\mathsf{env}_V \in \mathsf{Env}_V : \mathsf{VNames} \cup \mathsf{MVar} \rightharpoonup \mathsf{Val} \qquad \mathsf{env}_S \in \mathsf{Env}_S : \mathsf{ANames} \rightharpoonup \mathsf{Env}_F$$
$$\mathsf{env}_F \in \mathsf{Env}_F : \mathsf{FNames} \cup \{\mathtt{balance}\} \rightharpoonup \mathsf{Val} \qquad \mathsf{env}_T \in \mathsf{Env}_T : \mathsf{ANames} \rightharpoonup \mathsf{Env}_M$$
$$\mathsf{env}_M \in \mathsf{Env}_M : \mathsf{MNames} \rightharpoonup \mathsf{VNames}^* \times \mathsf{Stm}$$

We regard each environment env_J, for any $J \in \{V, F, M, S, T\}$, as a (finite) set of pairs (d, c) where $d \in \mathrm{dom}(\mathsf{env}_J)$ and $c \in \mathrm{codom}(\mathsf{env}_J)$. The notation $\mathsf{env}_J[d \mapsto c]$ denotes the update of env_J mapping d to c; environment updates are left-associative. We write $\mathsf{env}_J^{\emptyset}$ for the empty J-environment. To simplify the notation, when two or more environments appear together, we shall use the convention of writing the subscripts together (e.g., we use env_{ST} to stand for $\mathsf{env}_S, \mathsf{env}_T$). ■

Our binding model consists of two environments: a *method table* env_T, which maps addresses to method environments, and a *state* env_S, which maps addresses to lists of fields and their values.

$$[\textsc{decf}_1]\frac{}{\langle \epsilon, \mathsf{env}_F\rangle \to_{\mathrm{DF}} \mathsf{env}_F} \qquad [\textsc{decf}_2]\frac{\langle DF, \mathsf{env}_F\rangle \to_{\mathrm{DF}} \mathsf{env}'_F}{\langle \texttt{field } p \texttt{ := } v; DF, \mathsf{env}_F\rangle \to_{\mathrm{DF}} \{(p,v)\} \cup \mathsf{env}'_F}$$

$$[\textsc{decm}_1]\frac{}{\langle \epsilon, \mathsf{env}_M\rangle \to_{\mathrm{DM}} \mathsf{env}_M} \qquad [\textsc{decm}_2]\frac{\langle DM, \mathsf{env}_M\rangle \to_{\mathrm{DM}} \mathsf{env}'_M}{\langle f(\tilde{x})\ \{S\}\ DM, \mathsf{env}_M\rangle \to_{\mathrm{DM}} \{(f,(\tilde{x},S))\} \cup \mathsf{env}'_M}$$

$$[\textsc{decc}_1]\frac{}{\langle \epsilon, \mathsf{env}_{ST}\rangle \to_{\mathrm{DC}} \mathsf{env}_{ST}}$$

$$[\textsc{decc}_2]\frac{\left\langle DF, \mathsf{env}_F^{\emptyset}\right\rangle \to_{\mathrm{DF}} \mathsf{env}_F \quad \left\langle DM, \mathsf{env}_M^{\emptyset}\right\rangle \to_{\mathrm{DM}} \mathsf{env}_M \quad \langle DC, \mathsf{env}_{ST}\rangle \to_{\mathrm{DC}} \mathsf{env}'_{ST}}{\langle \texttt{contract } X\ \{DF\ \ DM\}\ DC, \mathsf{env}_{ST}\rangle \to_{\mathrm{DC}} \{(X, \mathsf{env}_F)\} \cup \mathsf{env}'_S, \{(X, \mathsf{env}_M)\} \cup \mathsf{env}'_T}$$

Fig. 4. Semantics of declarations.

Thus, for each contract, we have the list of methods it declares and its current state; of course, the method table is constant, once all declarations are performed, whereas the state will change during the evaluation of a program.

For example, if we consider the code given in Figure 1, we have that

$$\begin{array}{lcllcl} \mathsf{env}_S & = & \left\{(\mathtt{X}, \mathsf{env}_F^X), (\mathtt{Y}, \mathsf{env}_F^Y)\right\} & \mathsf{env}_T & = & \left\{(\mathtt{X}, \mathsf{env}_M^X), (\mathtt{Y}, \mathsf{env}_M^Y)\right\} \\ \mathsf{env}_F^X & = & \{(\texttt{called}, \mathtt{F}), \ldots\} & \mathsf{env}_M^X & = & \{(\texttt{transfer}, (y, \texttt{if } e \texttt{ then } S \texttt{ else skip})), \ldots\} \\ \mathsf{env}_F^Y & = & \{\ldots\} & \mathsf{env}_M^Y & = & \{(\texttt{deposit}, (x, x\texttt{.transfer()\$0})), \ldots\} \end{array} \tag{1}$$

where

$$\begin{array}{rcl} e & = & \neg\texttt{this.called} \wedge \texttt{this.balance} \geq 1 \\ S & = & S_1; S_2 \quad \text{with} \quad S_1 = y\texttt{.deposit(this)\$1} \quad \text{and} \quad S_2 = \texttt{this.called := T} \end{array}$$

2.2.1 Declarations. The semantics of the declarations builds the field and method environments env_F and env_M, and the state and method table env_S and env_T. We give the semantics in a classic big-step style and their defining rules are given in Figure 4. Without loss of generality, we assume that the field and method names are distinct within each contract, and that so are contract names; therefore, the rules in Figure 4 define finite partial functions.

2.2.2 Expressions. Figure 5 gives the semantics of expressions e. Expressions have no side effects, so they cannot contain method calls, but can access both local variables and fields of any contract. Thus, expression evaluations are of the form $\mathsf{env}_{SV} \vdash e \to_{\mathrm{e}} v$, i.e., they are relative to the state and variable environments. We use k to range over `this`, `sender`, `value` and variables x (i.e., $k \in \mathrm{dom}\,(\mathsf{env}_V)$), and q to range over `balance` and fields p (i.e., $q \in \mathrm{dom}\,(\mathsf{env}_F)$).

We do not give explicit rules for the boolean and integer operators subsumed under op, but simply assume that they can be evaluated to a *unique* value by some semantics $\mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v \in \mathbb{Z} \cup \mathbb{B}$.[2] It follows that each expression evaluates to a unique value relative to some given state and variable environments. Note that we assume that no operation can yield an address X; thus we disallow any form of pointer arithmetic.

2.2.3 Statements. The semantics of statements describes the actual execution steps of a program. In Figure 6, we give the semantics in big-step style, where a step describes the execution of a statement in its entirety. Statements can read from the method table and they can modify the state

[2]To simplify the definitions, we assume that all operations are total. Were this not the case, we would have needed some exception handling for partial operations (e.g., division by zero).

$$[\textsc{val}]\ \frac{}{\mathsf{env}_{SV} \vdash v \to_{\mathrm{e}} v}$$

$$[\textsc{var}]\ \frac{k \in \mathrm{dom}\,(\mathsf{env}_V) \quad \mathsf{env}_V(k) = v}{\mathsf{env}_{SV} \vdash k \to_{\mathrm{e}} v}$$

$$[\textsc{op}]\ \frac{\mathsf{env}_{SV} \vdash \tilde{e} \to_{\mathrm{e}} \tilde{v} \quad \mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v}{\mathsf{env}_{SV} \vdash \mathrm{op}(\tilde{e}) \to_{\mathrm{e}} v}$$

$$[\textsc{field}]\ \frac{\mathsf{env}_{SV} \vdash e \to_{\mathrm{e}} X \quad q \in \mathrm{dom}\,(\mathsf{env}_S(X)) \quad \mathsf{env}_S(X)(q) = v}{\mathsf{env}_{SV} \vdash e.q \to_{\mathrm{e}} v}$$

Fig. 5. Semantics of expressions.

(i.e., the variable and field bindings). The result of executing a statement is a new state, so transitions are of the form $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to_{\mathrm{s}} \mathsf{env}'_{SV}$ (recall that env'_{SV} stands for $\mathsf{env}'_S, \mathsf{env}'_V$), since both the field values in env_S and the values of the local variables in env_V may have been modified by the execution of S.

Most of the rules are straightforward; notice that we have no rule for the `throw` command, since this represents an abnormal termination of the program. The rule [DECV] is used when we declare a new variable x, with scope limited to the statement S; we implicitly assume alpha-conversion to handle the shadowing of an existing name. In the premise, we evaluate the expression e to a value v, and then execute the statement S with a variable environment where we have added the pair (x, v). During the execution of S, this variable environment may of course be updated (by applications of the rule [ASSV]), which may alter any value in the environment, including v. However, outside of the scope of the declaration, x is not visible and so the pair (x, v') is removed from the environment once S finishes. In contrast, any other change that led to env'_V (as well as any change that led to the global state env'_S) is retained.

The [CALL] rule is the most complicated, because we need to perform a number of actions:

- —First of all, we obviously have to evaluate the address and the parameters (viz., e_1, $\tilde{e}$ and e_2), relative to the current execution environment env_{SV}; we use the obtained address Y of the callee to retrieve the field environment env_F^Y for this contract and, through the method table, to extract the list of formal parameters $\tilde{x}$ and the body of the method S. We also have to verify that the number of actual parameters is the same as the number of formal parameters.
- —We have to check that the `balance` of the caller is at least z (that must be a non-negative number), and, in that case, update the state environment by subtracting z from the balance of X and adding z to the balance of Y, in their respective field environments; this yields a new state env''_S, where we write $\mathsf{env}_F[\texttt{balance -= } z]$ and $\mathsf{env}'_F[\texttt{balance += } z]$ for these two operations.
- —We build the new execution environment env''_V by creating new bindings for the special variables `this`, `sender` and `value`, and by binding the formal parameters $\tilde{x}$ to the values of the actual parameters $\tilde{v}$.
- —Finally, we execute the statement S in this new environment. This yields the new state env'_S and also an updated variable environment env'_V, since S may have modified the bindings in env''_V. However, these bindings are local to the method and therefore we discard them once the call finishes. So, the result of this transition is the updated state (env'_S) and the original variable environment of the caller (env_V).

$$[\textsc{skip}]\ \frac{}{\mathsf{env}_T \vdash \langle \mathtt{skip}, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}_{SV}}$$

$$[\textsc{seq}]\ \frac{\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}''_{SV} \quad \mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle S_1\,; S_2, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}}$$

$$[\textsc{if}]\ \frac{\mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} b \quad \mathsf{env}_T \vdash \langle S_b, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \mathtt{if}\ e\ \mathtt{then}\ S_{\mathsf{T}}\ \mathtt{else}\ S_{\mathsf{F}}, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}}\ (b \in \{\mathsf{T}, \mathsf{F}\})$$

$$[\textsc{while}_{\mathsf{T}}]\ \frac{\mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} \mathsf{T} \qquad \mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}''_{SV} \quad \mathsf{env}_T \vdash \langle \mathtt{while}\ e\ \mathtt{do}\ S, \mathsf{env}''_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \mathtt{while}\ e\ \mathtt{do}\ S, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}}$$

$$[\textsc{while}_{\mathsf{F}}]\ \frac{\mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} \mathsf{F}}{\mathsf{env}_T \vdash \langle \mathtt{while}\ e\ \mathtt{do}\ S, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}_{SV}}$$

$$[\textsc{decv}]\ \frac{x \notin \mathrm{dom}\,(\mathsf{env}_V) \qquad \mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} v \quad \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, \{(x,v)\} \cup \mathsf{env}_V\rangle \to_{\mathsf{S}} \mathsf{env}'_S\,, \{(x,v')\} \cup \mathsf{env}'_V}{\mathsf{env}_T \vdash \langle \mathtt{var}\ x\ \mathtt{:=}\ e\ \mathtt{in}\ S,\ \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}}$$

$$[\textsc{assv}]\ \frac{x \in \mathrm{dom}\,(\mathsf{env}_V) \qquad \mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} v}{\mathsf{env}_T \vdash \langle x\ \mathtt{:=}\ e, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}_S, \mathsf{env}_V[x \mapsto v]}$$

$$[\textsc{assf}]\ \frac{\mathsf{env}_V(\mathtt{this}) = X \quad \mathsf{env}_S(X) = \mathsf{env}_F \quad p \in \mathrm{dom}\,(\mathsf{env}_F) \quad \mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} v}{\mathsf{env}_T \vdash \langle \mathtt{this}.p\ \mathtt{:=}\ e, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}_S[X \mapsto \mathsf{env}_F[p \mapsto v]]\,, \mathsf{env}_V}$$

$$[\textsc{call}]\ \frac{\begin{array}{l}\mathsf{env}_{SV} \vdash e_1 \to_{\mathsf{e}} Y \qquad \mathsf{env}_S(Y) = \mathsf{env}^Y_F \qquad (\mathsf{env}_T(Y))(f) = (\tilde{x}, S)\\ |\tilde{x}| = |\tilde{e}| = h \qquad \mathsf{env}_{SV} \vdash \tilde{e} \to_{\mathsf{e}} \tilde{v} \qquad \mathsf{env}_{SV} \vdash e_2 \to_{\mathsf{e}} z\\ \mathsf{env}_V(\mathtt{this}) = X \qquad \mathsf{env}_S(X) = \mathsf{env}^X_F \qquad 0 \le z \le \mathsf{env}^X_F(\mathtt{balance})\\ \mathsf{env}''_S = \mathsf{env}_S[X \mapsto \mathsf{env}^X_F[\mathtt{balance\ -=}\ z]][Y \mapsto \mathsf{env}^Y_F[\mathtt{balance\ +=}\ z]]\\ \mathsf{env}''_V = \{(\mathtt{this}, Y), (\mathtt{sender}, X), (\mathtt{value}, z), (x_1, v_1), \ldots, (x_h, v_h)\}\\ \mathsf{env}_T \vdash \langle S, \mathsf{env}''_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle e_1\,.\,f(\tilde{e})\$e_2, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_S, \mathsf{env}_V}$$

$$[\textsc{dcall}]\ \frac{\begin{array}{l}\mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} Y \qquad \mathsf{env}_T(Y)(f) = (\tilde{x}, S) \qquad |\tilde{x}| = |\tilde{e}| = h \qquad \mathsf{env}_{SV} \vdash \tilde{e} \to_{\mathsf{e}} \tilde{v}\\ \mathsf{env}''_V = \{(\mathtt{this}, \mathsf{env}_V(\mathtt{this})), (\mathtt{sender}, \mathsf{env}_V(\mathtt{sender})), (\mathtt{value}, \mathsf{env}_V(\mathtt{value})),\\ \qquad (x_1, v_1), \ldots, (x_h, v_h)\}\\ \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, \mathsf{env}''_V\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}\end{array}}{\mathsf{env}_T \vdash \langle \mathtt{dcall}\ e\,.\,f(\tilde{e}), \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_S, \mathsf{env}_V}$$

Fig. 6. Big-step semantics of statements in TinySol.

It should be noted that a *local* method call, i.e., a call to a method within the same (calling) contract, is merely a special case of the rule [call]. Local calls would have the form $\mathtt{this}.f(\tilde{e})\$0$, as transferring any amount of currency will not alter the balance of the contract.

Example 1. Consider the code in Figure 1 and the associated environments, provided in (1). The execution of the method invocation `X.transfer(Y)$0` done by some user A relies on rule [CALL]. By its premises, we have to retrieve the code of method `transfer` associated to `X` in env_T, i.e., we have to consider $\mathsf{env}_T(\mathtt{X})(\mathtt{transfer})$ and take the second element of the pair $(y, \mathtt{if}\ e\ \mathtt{then}\ S\ \mathtt{else\ skip})$, and run it under the environment

$$\mathsf{env}_V = \{(\mathtt{this}, \mathtt{X}), (\mathtt{sender}, A), (\mathtt{value}, 0), (y, \mathtt{Y})\}$$

(indeed, since there is no currency exchange in this invocation, the state env_S does not change—and no check is needed on the balance of A). By recalling that

$$\begin{array}{rcl} e & = & \neg\mathtt{this.called} \wedge \mathtt{this.balance} \geq 1 \\ S & = & S_1; S_2 \quad \text{where} \quad S_1 = y.\mathtt{deposit(this)\$1} \quad \text{and} \quad S_2 = \mathtt{this.called := T} \end{array}$$

we have to calculate the final state for

$$\mathsf{env}_T \vdash \langle \mathtt{if}\ e\ \mathtt{then}\ S\ \mathtt{else\ skip}, \mathsf{env}_{SV} \rangle \tag{2}$$

To this end, we first have to evaluate e under the current env_{SV}: assuming that $\mathsf{env}_S(\mathtt{X})(\mathtt{balance})$ is positive, since $\mathsf{env}_S(\mathtt{X})(\mathtt{called}) = \mathtt{F}$, the guard evaluates to `T` and we are required to evaluate the code of the then-branch. Since this is a sequential composition, we have to first evaluate the subcommand S_1, that is again a method call, handled through [CALL]. Since $\mathsf{env}_V(y) = \mathtt{Y}$ and $\mathsf{env}_V(\mathtt{this}) = \mathtt{X}$, we have to evaluate `Y.deposit(X)$1`. Since $\mathsf{env}_T(\mathtt{Y})(\mathtt{deposit}) = (x, x.\mathtt{transfer(this)\$0})$, we have to evaluate $x.\mathtt{transfer(this)\$0}$ in the environments

$$\begin{array}{rcl} \mathsf{env}'_V & = & \{(\mathtt{this}, \mathtt{Y}), (\mathtt{sender}, X), (\mathtt{value}, 1), (x, \mathtt{X})\} \\ \mathsf{env}'_S & = & \mathsf{env}_S[X \mapsto \mathsf{env}_F^X[\mathtt{balance\ -=\ 1}]][Y \mapsto \mathsf{env}_F^Y[\mathtt{balance\ +=\ 1}]] \end{array}$$

that is, we have to calculate the final state for

$$\mathsf{env}_T \vdash \langle x.\mathtt{transfer(this)\$0}, \mathsf{env}'_{SV} \rangle$$

Since $\mathsf{env}'_V(x) = \mathtt{X}$ and $\mathsf{env}'_V(\mathtt{this}) = \mathtt{Y}$, we are back to evaluating `X.transfer(Y)$0`, this time invoked by contract `Y` and with a balance of `X` decreased by 1 unit. Now, the above reasoning is repeated until all balance of `X` is totally transferred to `Y`, as we informally argued in the introduction section; let $\widehat{\mathsf{env}_S}$ be the resulting state (where all $\mathsf{env}_S(\mathtt{X})(\mathtt{balance})$ is moved into $\mathsf{env}_S(\mathtt{Y})(\mathtt{balance})$). At that moment, the premise of the if-then-else becomes `F`, since $\widehat{\mathsf{env}_S}(\mathtt{X})(\mathtt{balance}) = 0$. This completes the execution of S_1 in (2) and enables the execution of S_2, which modifies $\widehat{\mathsf{env}_S}$ by putting `T` into `X.called`. This completes the execution of `X.transfer(Y)$0` done by A. ∎

Finally, rule [DCALL] accounts for delegate calls; there are a few things to note about the semantics of this construct. First, the method body S is executed in the context of the caller; i.e., the magic variables `this`, `sender` and `value` are *not* rebound. In particular, this means that any access to fields (through `this`) are to fields in the *caller* contract, rather than in the contract Y (where f is defined). Second, there is no value associated to the call, so funds cannot be transferred to the callee with this type of call. This is consistent with the idea that the code is executed in the context of the caller; i.e., *as if* the method had been declared in the caller contract (and so its effect is similar to that of a local call, i.e., the `balance` is unchanged).

2.2.4 Transactions and Blockchains. The semantics for blockchains is given as a transition system defined by the rules given in Figure 7. Here, the rule [GENESIS] describes the 'genesis event' where contracts are declared, whilst rules [REVELATION] and [TRANS] describe the execution of a transaction. This is thus a *small-step* semantics, invoking the big-step semantics for declarations and statements in the premises of its rules. We denote by $\rightarrow_{\mathrm{B}}^{+}$ the transitive closure of $\rightarrow_{\mathrm{B}}$. We

$$[\textsc{Genesis}]\ \frac{\left\langle DC, \mathsf{env}_{ST}^{\emptyset}\right\rangle \to_{\mathrm{DC}} \mathsf{env}_{ST}}{\left\langle DC\ \ \tau, \mathsf{env}_{ST}^{\emptyset}\right\rangle \to_{\mathrm{B}} \langle \tau, \mathsf{env}_{ST}\rangle} \qquad [\textsc{Revelation}]\ \frac{}{\langle \epsilon, \mathsf{env}_{ST}\rangle \to_{\mathrm{B}} \mathsf{env}_{ST}}$$

$$[\textsc{Trans}]\ \frac{\mathsf{env}_T \vdash \langle X\,.\,f(\tilde{v})\$z, \mathsf{env}_S, \{(\mathtt{this}, A)\}\rangle \to_{\mathrm{S}} \mathsf{env}'_S, \mathsf{env}_V}{\left\langle A \overset{z}{\rightsquigarrow} X\,.\,f(\tilde{v}), \tau, \mathsf{env}_{ST}\right\rangle \to_{\mathrm{B}} \langle \tau, \mathsf{env}'_S, \mathsf{env}_T\rangle}$$

Fig. 7. Semantics of blockchains.

remark that the rules of the operational semantics for blockchains (as well as those for statements presented above) define a *deterministic* transition relation.

Note that, unlike in the original formulation of TinySol, we do not include a rule like [Tx2] in [7] for rolling back a transaction in case it is non-terminating or it aborts via a `throw` command. Such a rule would require a premise that cannot be checked effectively for a Turing-complete language like TinySol and therefore we omit it, since it is immaterial for the main contributions we give in this article[3]. In practice, termination of Ethereum smart contracts is ensured via a 'gas mechanism' and is assumed by techniques for the formal analysis of smart contracts (see, e.g., [2]). Intuitively, the gas cost of a transaction reflects the amount of computational work required for completing it. Whenever a user schedules a transaction, two additional parameters have to be specified: the *gas limit* (an upper bound on the gas cost of the transaction—if this limit is exceeded, an *out-of-gas exception* is thrown and the transaction aborts) and the *gas price* (the amount of currency that the user is willing to pay for executing the transaction). This ensures that users will not schedule long-running transactions and just pick an arbitrarily high gas limit, since the gas price will still be paid for each unit of gas consumed in a transaction, even if the transaction aborts with an out-of-gas exception. However, as observed in, for instance, [22], proof of termination for smart contracts is non-trivial even in the presence of a 'gas mechanism'. In the aforementioned paper, the authors present the first mechanised proof of termination of contracts written in **Ethereum virtual machine (EVM)** bytecode using minimal assumptions on the gas cost of operations (see the study [54] for an empirical analysis of the effectiveness of the 'gas mechanism' in estimating the computational cost of executing real-life transactions). We leave for future work the adaptation of the results we present in this article to the setting of TinySol with the 'gas mechanism' presented in [2]. This would complicate both the operational semantics (by adding the checks that verify the presence of enough gas for executing every method operation and the associated update of the total amount of gas) and the type system (that somehow has to verify that the specified amount of gas is enough and no out-of-gas exception is raised).

3 Call Integrity and Noninterference in TinySol

Grishchenko et al. [26] formulate the property of *call integrity* for smart contracts written in the language EVM, which is the 'low-level' bytecode of the Ethereum platform, and the target language to which, e.g., Solidity compiles. They then prove [26, Theorem 1] that this property suffices to rule out reentrancy phenomena, as those described in the example in Figure 1. We first formulate a similar property for TinySol; this requires a few preliminary definitions.

Definition 3 *(Trace Semantics).* A *trace* of method invocations is given by

$$\pi ::= \epsilon \mid X \overset{z}{\rightsquigarrow} Y\,.\,f(\tilde{v}), \pi$$

[3]For instance, rule [Tx2] in [7] has an undecidable premise that checks whether the execution of the body of a contract does *not* yield a final state. It is debatable whether such rules should appear in an operational semantics.

where X is the address of the calling contract, Y is the address of the called contract, f is the method name, and $\tilde{v}$ and z are the actual parameters. We annotate the big-step semantics with a trace containing information on the invoked methods to yield labelled transitions of the form $\xrightarrow{\pi}_{\mathrm{S}}$. To do this, we modify the rules in Figure 6 as follows:

—in rules [SKIP], [WHILE$_{\mathrm{F}}$], [ASSV] and [ASSF], every occurrence of $\rightarrow_{\mathrm{S}}$ becomes $\xrightarrow{\epsilon}_{\mathrm{S}}$;
—in rules [IF] and [DECV], every occurrence of $\rightarrow_{\mathrm{S}}$ becomes $\xrightarrow{\pi}_{\mathrm{S}}$;
—rules [SEQ], [WHILE$_{\mathrm{T}}$], [CALL] and [DCALL] respectively become:

$$\frac{\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1}_{\mathrm{S}} \mathsf{env}''_{SV} \qquad \mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \xrightarrow{\pi_2}_{\mathrm{S}} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle S_1\,;S_2, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1,\pi_2}_{\mathrm{S}} \mathsf{env}'_{SV}}$$

$$\frac{\mathsf{env}_{SV} \vdash e \rightarrow_{\mathrm{e}} \mathsf{T} \qquad \mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1}_{\mathrm{S}} \mathsf{env}''_{SV} \quad \mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}''_{SV}\rangle \xrightarrow{\pi_2}_{\mathrm{S}} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \texttt{while}\ e\ \texttt{do}\ S, \mathsf{env}_{SV}\rangle \xrightarrow{\pi_1,\pi_2}_{\mathrm{S}} \mathsf{env}'_{SV}}$$

$$\frac{\ldots \qquad \mathsf{env}_T \vdash \langle S, \mathsf{env}''_{SV}\rangle \xrightarrow{\pi}_{\mathrm{S}} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle e_1.f(\tilde{e})\$e_2, \mathsf{env}_{SV}\rangle \xrightarrow{X \overset{z}{\rightsquigarrow} Y.f(\tilde{v}),\pi}_{\mathrm{S}} \mathsf{env}'_S, \mathsf{env}_V}$$

$$\frac{\ldots \qquad \mathsf{env}_V(\texttt{this}) = X \qquad \mathsf{env}_T \vdash \langle S, \mathsf{env}_S, \mathsf{env}''_V\rangle \xrightarrow{\pi}_{\mathrm{S}} \mathsf{env}'_{SV}}{\mathsf{env}_T \vdash \langle \texttt{dcall}\ e.f(\tilde{e}), \mathsf{env}_{SV}\rangle \xrightarrow{X \overset{0}{\rightsquigarrow} Y.f(\tilde{v}),\pi}_{\mathrm{S}} \mathsf{env}'_S, \mathsf{env}_V}$$

We extend this annotation to the semantics for blockchains, written $\xrightarrow{\pi}_{\mathrm{B}}$, and defined such that rule [REVELATION] generates $\xrightarrow{\epsilon}_{\mathrm{B}}$ and rule [TRANS] now becomes

$$\frac{\mathsf{env}_T \vdash \langle X.f(\tilde{v})\$z, \mathsf{env}_S, \{(\texttt{this}, A)\}\rangle \xrightarrow{X \overset{z}{\rightsquigarrow} Y.f(\tilde{v})}_{\mathrm{S}} \mathsf{env}'_S, \mathsf{env}_V}{\left\langle A \overset{z}{\rightsquigarrow} X.f(\tilde{v}), \tau, \mathsf{env}_{ST}\right\rangle \xrightarrow{X \overset{z}{\rightsquigarrow} Y.f(\tilde{v})}_{\mathrm{B}} \langle \tau, \mathsf{env}'_S, \mathsf{env}_T\rangle}$$

The relation $\xrightarrow{\pi}_{\mathrm{B}}$ is obtained by composing the transitions labelled by the method invocations in π—that is, we write $\langle T_1, \mathsf{env}^1_{ST}\rangle \xrightarrow{\pi_1,\ldots,\pi_h}_{\mathrm{B}} \mathsf{env}^{h+1}_{ST}$ whenever $\langle T_1, \mathsf{env}^1_{ST}\rangle \xrightarrow{\pi_1}_{\mathrm{B}} \ldots \langle T_h, \mathsf{env}^h_{ST}\rangle \xrightarrow{\pi_h}_{\mathrm{B}} \mathsf{env}^{h+1}_{ST}$, where $\pi_i = X_i \overset{z_i}{\rightsquigarrow} Y_i.f_i(\widetilde{v_i})$ for each $i \in \{1, \ldots, h\}$. Notice that $\langle T, \mathsf{env}_{ST}\rangle \rightarrow^+_{\mathrm{B}} \mathsf{env}'_{ST}$ if, and only if, $\langle T, \mathsf{env}_{ST}\rangle \xrightarrow{\pi}_{\mathrm{B}} \mathsf{env}'_{ST}$ for some π. ■

Definition 4 *(Projection).* The *projection* of a trace to a specific contract X, written $\pi \downarrow_X$, is the trace of calls with X as the calling address. Formally:

$$\epsilon \downarrow_X = \epsilon \qquad (Z \overset{z}{\rightsquigarrow} Y.f(\tilde{v}), \pi) \downarrow_X = \begin{cases} X \overset{z}{\rightsquigarrow} Y.f(\tilde{v}), (\pi \downarrow_X) & \text{if } Z = X \\ \pi \downarrow_X & \text{otherwise} \end{cases}$$ ■

Notationally, given a (partial) function f, we write $f|_{\mathcal{X}}$ to denote the restriction of f to the subset $\mathcal{X}$ of its domain.

Definition 5 *(Call Integrity).* Let ANames be the set of all contracts (addresses), $\mathcal{X} \subseteq$ ANames denote a set of trusted contracts and $\mathcal{Y} \triangleq$ ANames $\setminus\, \mathcal{X}$ stand for all other contracts. A contract $X \in \mathcal{X}$ has *call integrity* for $\mathcal{Y}$ if, for every transaction τ and environments env^1_{ST} and env^2_{ST} such that

$\mathsf{env}^1_{ST}|_{\mathcal{X}} = \mathsf{env}^2_{ST}|_{\mathcal{X}}$, it holds that

$$\langle \tau, \mathsf{env}^1_{ST} \rangle \xrightarrow{\pi_1}_{\mathrm{B}} \mathsf{env}^{1'}_{ST} \wedge \langle \tau, \mathsf{env}^2_{ST} \rangle \xrightarrow{\pi_2}_{\mathrm{B}} \mathsf{env}^{2'}_{ST} \implies \pi_1{\downarrow_X} = \pi_2{\downarrow_X}$$ ■

The definition is quite complicated and contains a number of elements:

- X is the contract of interest.
- $\mathcal{X}$ is a set of *trusted* contracts, which we assume are allowed to influence the behaviour of X. This set must obviously contain X, since X must at least be assumed to be trusted. Thus, a contract X can have call integrity for all contracts if $\mathcal{X} = \{ X \}$.
- Conversely, the set $\mathcal{Y} = \mathsf{ANames} \setminus \mathcal{X}$ is the set of addresses of all contracts that are *untrusted*[4].
- env^1_{ST} and env^2_{ST} are any two pairs of method/field environments that coincide (both in the code and in the values) for all the trusted contracts[5].
- τ is any transaction; it may be issued from any account and to any contract.

The point is that the contracts in $\mathcal{X}$ are assumed to be known, and hence invariant in the environments env^1_{ST} and env^2_{ST}, whereas any contract in $\mathcal{Y}$ is assumed to be unknown and may be controlled by an attacker (this is similar to the threat model assumed in [24]); moreover, also the issued transaction is part of the execution context. Hence, as usual, the attacker model adopted by Grishchenko et al. [26] in defining call integrity assumes that the attacker fully controls the execution context; this is the reason for quantifying over all possible transactions and contract declarations where the contracts in $\mathcal{X}$ can be run. So, the call integrity property intuitively requires that, if we run the trusted part of the code in any execution context, then the behaviour of X remains the same, i.e., X must make exactly the same method calls (and in exactly the same order), irrespectively of what the attacker can do to influence X's behaviour. Thus, to *disprove* that X has call integrity, it suffices to find two environments and a transaction that will induce a difference in the call trace of X, i.e., a possible attacker that is able to lead X to different (and possibly unwanted) behaviours.

The idea underlying call integrity is that the behaviour of X should not depend on any untrusted code (i.e., contracts in $\mathcal{Y}$), even if control is transferred to a contract in $\mathcal{Y}$. The latter could, for example, happen if X_1 calls a method on $X_2 \in \mathcal{X}$, and X_2 then calls a method on a contract in $\mathcal{Y}$. This also means that X cannot directly call any contract in $\mathcal{Y}$. For example, consider a transaction that contains `X.f(Y)$0`; it can then be possible that X calls a method on the contract whose address is received as a parameter, like in the case in which `X` contains a method `f` such that

```
f(y) {y.g()$0}
```

Another possibility is if X calls a method on a 'hard-coded' contract address, for example with a transaction that contains `X.f()$0` and `X` contains a method `f` such that

```
f() {Y.g()$0}
```

In both cases, we can easily pick up two environments that are able to induce different behaviours: for example, by choosing env^1_{ST} and env^2_{ST} such that $\mathtt{Y} \in \mathrm{dom}\left(\mathsf{env}^1_T\right)$ and $\mathtt{Y} \notin \mathrm{dom}\left(\mathsf{env}^2_T\right)$. This can seem somewhat contrived, especially if we assume that all contracts are created at the genesis event, and it might therefore be reasonable to require also that $\mathrm{dom}\left(\mathsf{env}^1_{ST}\right) = \mathrm{dom}\left(\mathsf{env}^2_{ST}\right)$, so that we at least assume that contracts exist at the same addresses. However, on an actual blockchain,

[4]Note that this is formulated inversely by Grishchenko, Maffei and Schneidewind who instead formulate the property for a set of *untrusted* contracts $\mathcal{A}_{\mathcal{C}}$, corresponding to $\mathcal{Y}$ in the present formulation. However, using the set of *trusted* addresses $\mathcal{X}$ seems more straightforward.

[5]This too is inversely formulated by Grishchenko, Maffei and Schneidewind.

new contracts can be deployed (and in some cases also deleted) at any time, and if such a degree of realism is desired, this extra constraint should not be imposed.

Remark 1. We remark here that a delegate call is recorded in the trace exactly as if it was a normal call (see the cases for [CALL] and [DCALL] in Definition 3). This is what is prescribed in [26] for defining call integrity. Since delegate calls do not modify the balance fields (but only execute the code in the context of the caller), we believe that we could avoid including delegate calls in the traces and still obtaining a notion of 'call-integrity' that would prevent reentrancy, since the control flow does not leave the caller contract. However, we prefer to remain faithful to the original definition of [26] and to the associated proof that call-integrity implies non-reentrancy. ■

The main problem with the definition of call integrity is that it relies on a universal quantification over all possible execution contexts. This makes it difficult to check that property in practice. However, our previous discussion indicates that call integrity may intuitively be viewed as a form of *noninterference* between the trusted and the untrusted contracts. We now see to what extent this intuition is true and formally compare the two notions.

First of all, we consider a basic lattice of security levels, made up by just two levels, namely H (for *high*) and L (for *low*), with $L \sqsubseteq H$. We label every contract as high or low through a contracts-to-levels mapping $\lambda : \mathsf{ANames} \to \{L, H\}$; this induces a bipartition of the contract names ANames into the following sets:

$$\mathcal{L} = \{\, X \in \mathsf{ANames} \mid \lambda(X) = L \,\} \qquad \mathcal{H} = \{\, X \in \mathsf{ANames} \mid \lambda(X) = H \,\}$$

In this way, we create a bipartition of the state into low and high, corresponding to the fields of the low and high contracts, respectively. Then, we define *low-equivalence* $=_L$ to be the equivalence on states such that $\mathsf{env}_S^1 =_L \mathsf{env}_S^2$ if and only if $\mathsf{env}_S^1(X) = \mathsf{env}_S^2(X)$, for every $X \in \mathcal{L}$.

We can now adapt the notion of noninterference for multi-threaded programs by Smith and Volpano [47] to the setting of TinySol.

Definition 6 *(Noninterference).* Given a contracts-to-levels mapping $\lambda : \mathsf{ANames} \to \{L, H\}$ and a contract environment env_T, the contracts satisfy *noninterference* if, for every env_S^1 and env_S^2 and for every transaction τ such that

$$\mathsf{env}_S^1 =_L \mathsf{env}_S^2 \qquad \langle \tau, \mathsf{env}_S^1, \mathsf{env}_T \rangle \to_{\mathrm{B}}^{+} \mathsf{env}_S^{1'}, \mathsf{env}_T \qquad \langle \tau, \mathsf{env}_S^2, \mathsf{env}_T \rangle \to_{\mathrm{B}}^{+} \mathsf{env}_S^{2'}, \mathsf{env}_T$$

it holds that $\mathsf{env}_S^{1'} =_L \mathsf{env}_S^{2'}$. ■

As usual, noninterference can be used both for *secrecy* and *integrity*. In the first case, H denotes the secret data that should not leak any piece of information to the public data L. The second case is dual: the trusted data is labelled with L, and should not receive any flow from the untrusted H data. In this second case, the names high/low can be thought of as speaking about the level of *distrust*. The attacker model underlying noninterference is the standard one (see, e.g., [4, 5]): the attacker has full knowledge of the low memory, so assignments to low fields are the only things that an attacker can observe. Noninterference requires that the attacker cannot gain any information on the high part of the memory, regardless of the transaction it performs. (As for call integrity, executing a transaction is the only way in which an attacker can observe the high memory.) In particular, whatever the high values are, the low memory locations remain equal, if they were equal at the outset.

Remark 2 *(Incomparability).* Call integrity and noninterference seem strongly related, in the sense that the first requires that the behaviour of a contract is not influenced by the (bad) execution context, whereas the second one requires that a part of the computation (the 'low' one) is not influenced by the remainder context (the 'high' one). So, one may try to prove a statement like:

'$X \in \mathcal{X}$ has call integrity for $\mathcal{Y} \triangleq \mathsf{ANames} \setminus \mathcal{X}$ if and only if it satisfies noninterference w.r.t. λ such that $\mathcal{L} = \mathcal{X}$ and $\mathcal{H} = \mathcal{Y}$'. However, both directions are false.

To prove that call integrity does not imply noninterference, consider the following two contracts:

```
contract X {
  field balance = 0;
  send() { skip }
  go() { skip }
}
```

```
contract Y {
  field balance := v;
  send() { skip }
  go() { X.go()$this.balance }
}
```

where X is trusted and Y untrusted. Since X cannot invoke any method, this example satisfies call integrity. However, it does not satisfy noninterference. To see this, consider two environments, one assigning 1 to Y's balance and the other one assigning 0, and the transaction $\mathtt{Y} \overset{0}{\rightsquigarrow} \mathtt{Y.go()}$.

To prove that noninterference does not imply call integrity, consider the following contracts:

```
contract X {
  field balance := 0;
  send() { skip }
  go() {
      if Y.balance = 0
      then Z.a()$0
      else Z.b()$0
  }
}
```

```
contract Y {
  field balance := v;
  send() { skip }
}
```

```
contract Z {
  field balance := 0;
  send() { skip }
  a() { skip }
  b() { skip }
}
```

Assuming that both X and Z are low (i.e., trusted), the example satisfies noninterference: there is no way for Y (i.e., the untrusted contract) to influence the low (trusted) memory. By contrast, the code does not satisfy call integrity. Indeed, let v be 0 in one environment and 1 in the other, and consider τ to be $\mathtt{X} \overset{0}{\rightsquigarrow} \mathtt{X.go()}$: in the first environment, it generates $\mathtt{X} \overset{0}{\rightsquigarrow} \mathtt{Z.a()}$, whereas in the second one it generates $\mathtt{X} \overset{0}{\rightsquigarrow} \mathtt{Z.b()}$. ■

4 A Type System for Noninterference and Call Integrity

As demonstrated in Remark 2, call integrity and noninterference are incomparable properties. This is so because noninterference is a 2-property on the pair of *stores* $(\mathsf{env}_S^{1'}, \mathsf{env}_S^{2'})$ resulting from two different executions, whereas call integrity is a 2-property on the pair of *call traces* (π_1, π_2) generated during two executions. However, the two properties have an interesting overlap, because an outgoing currency flow (i.e., a method call) may also result, at least potentially, in a change of the stored values of the balance fields of the sender and of the recipient. Therefore, from a typing perspective, every method call yields an information flow between the two contracts involved, even when no amount of currency is actually transferred. Since a type system cannot have any information about the value that the expression representing the currency transferred will assume at runtime, to be on the safe side, it must always assume that this is a non-zero value. In [53], Volpano et al. devise a type system for checking information flows, which, as they show, yields a sound approximation to noninterference. In what follows, we create an adaptation of this type system tailored for TinySol and show that it may *also* be used to soundly approximate call integrity.

Overview. Since the type system has many ingredients, for the sake of clarity, we find it useful to present them briefly at a high level before delving into details in the subsequent subsections. First of all, types carry classic typing information (such as the type of data), as well as a security-related aspect, since they are supposed to describe allowed flows of information within program

fragments. We define the language of types in Definition 7, which, besides the conventional base types int and bool for integer and boolean values, also includes a notion of 'interface' to describe the type of an address/contract, consisting of the signatures of the methods and fields of the contract (Definition 8).

For the purpose of building a subtyping relation, interfaces must declare how they relate to each other. Specifically, we require that each interface must declare a single super-type, so that interfaces form a rooted 'inheritance tree', with a special interface $I^\top$ as the root. This interface, corresponding to the most basic implementation of a contract, is thus the common super-type of all other interfaces. We include a sanity check of the declared interfaces in Figure 8, which ensures that it indeed has the appropriate structure (a hierarchy with $I^\top$ as the top-most element), and that each member of each super-type interface also is a super-type of the corresponding member in each of its subtype interfaces. This subtyping relation for interface members (i.e., fields and methods) is defined in Figure 9. We separate it from the general subtyping relation for expression types, given in Figure 10, because this relationship only needs to be verified once.

The final component of the type system is the type rules for expressions, statements, environments and declarations. These are given in Figures 11–16.

4.1 Types, Type Environments and Typed Syntax

We begin by assuming a finite lattice $(\mathcal{S}, \sqsubseteq)$ consisting of a set of *security levels* $\mathcal{S}$, ranged over by s, and equipped with a partial order $\sqsubseteq$ with $s_\bot$ and $s_\top$ as least and largest elements. In the simplest setting, we have $\mathcal{S} \triangleq \{ L, H \}$ (for 'low' and 'high') and $L \sqsubseteq H$. This is sufficient for ensuring bipartite noninterference, but the type system can also handle more fine-grained security control.

Besides the security-level aspect, we shall also need a conventional data-type aspect of our types, so that we can assign types to addresses representing the contract residing at that address. For this, we shall borrow from conventional object-oriented type systems, such as the classic one presented in [30], similar to the type system for **Featherweight Solidity (FS)** given by Crafa et al. in [16]. Specifically, we let the type of an address be what we call an *interface* I; interfaces are defined in Definition 8 and specify the signatures of the fields and methods of the contract residing at that address. To give our types a uniform format, we also include int and bool as base types, and therefore let a data type be a pair (B, s) consisting of a base type B and a security level s. We shall use the notations (B, s) and B_s interchangeably, preferring the latter except when we need indices on types. The types are as follows.

Definition 7 *(Syntax of Types and Environments).* We use the following types and environments:

$$
\begin{aligned}
B \in \mathcal{B} &::= \mathsf{int} \mid \mathsf{bool} \mid I \\
T \in \mathcal{T} &::= B_s \mid \mathsf{var}(B_s) \mid \mathsf{cmd}(s) \mid \mathsf{proc}(\widetilde{B_s}) : s' \\
n \in \mathcal{N} &::= \mathsf{VNames} \cup \mathsf{FNames} \cup \mathsf{MNames} \cup \mathsf{ANames} \cup \mathsf{INames} \\
&\qquad \cup \{\mathtt{balance}, \mathtt{this}, \mathtt{sender}, \mathtt{value}\} \\
\Gamma, \Delta \in \mathcal{E} &::= \mathcal{N} \rightharpoonup \mathcal{T} \cup \mathcal{E} \\
\Sigma &::= \mathsf{INames} \rightharpoonup \mathsf{INames}
\end{aligned}
$$

where INames are interface names, ranged over by I. We write $\widetilde{T}$ for a tuple of types $(T_1, ..., T_h)$. ■

The meaning of the types is as follows:

—The type $\mathsf{var}(B_s)$ is given to containers (i.e., variables x or fields p) capable of storing a value of type B of security level s *or lower*.

$$[\Sigma\text{-TOP}]\frac{}{\Sigma;\Gamma \vdash (I^\top, I^\top)}$$

$$[\Sigma\text{-REC}]\frac{\begin{array}{ccc} I_1 \neq I_2 & \nexists I'.\Sigma'(I') = I_1 & \Gamma(I_1) = \Gamma_1 \qquad \Gamma(I_2) = \Gamma_2 \\ \mathrm{dom}\,(\Gamma_2) \subseteq \mathrm{dom}\,(\Gamma_1) & \forall n \in \mathrm{dom}\,(\Gamma_2).\Sigma \vdash \Gamma_1(n) <: \Gamma_2(n) & \Sigma;\Gamma \vdash \Sigma' \end{array}}{\Sigma;\Gamma \vdash (I_1, I_2), \Sigma'}$$

Fig. 8. Consistency rules for Σ.

- The type B_s is given to expressions e such that all accesses to containers appearing in e that are needed to evaluate it (here considered as '*reads*') are from containers of security level s *or lower*, and the resulting value is of type B.
- $\mathsf{cmd}(s)$ is a *phrase type* given commands S. It denotes that all *assignments* in the code are made to containers whose security level is s *or higher*.
- The type $\mathsf{proc}(\widetilde{B_s}) : s'$ is given to methods f. It denotes that the body S can be typed as $\mathsf{cmd}(s')$, under the assumption that the formal parameters $\tilde{x}$ have types $\widetilde{\mathsf{var}(B_s)}$ (i.e., a sequence of types $\mathsf{var}(B_1, s_1), \ldots, \mathsf{var}(B_h, s_h)$). Note that every method declaration contains an implicit write to the `balance` field of the containing contract: hence, given the meaning of $\mathsf{cmd}(s')$, this also means that the security level of `balance` must always be equal to or higher than the level of every method declared in an interface (s' in this case). For instance, if an interface specifies the signature of two methods with types $\mathsf{proc}(\mathsf{var}(B_1, s_1)) : s_1'$ and $\mathsf{proc}(\mathsf{var}(B_2, s_2)) : s_2'$, then the security level of `balance` must be equal to, or higher than, the least upper bound of s_1' and s_2'.

The meaning of the base types B is mostly straightforward: for integer and boolean values, we assume that the base type can be determined directly by observing the value itself; i.e., 5 is an integer and F is a boolean. However, with addresses $X \in \mathsf{ANames}$, the situation is different, as there is no inherent relationship between the address itself and its type, since addresses are *declared* to have a type (and a security level), according to the contract located at the address. The type of an address X is therefore an *interface* I, consisting of the signatures of the methods and fields of that contract. The interface could in principle be extracted from the contract definition itself; however, here we prefer to let interfaces be defined separately, since doing so will allow multiple contracts to implement the same interface. Interfaces are used both in the type system and to declare a tree-like hierarchy of contracts (similar to the class hierarchy of Java), mirroring the syntax of contract declarations (hence, also here we assume that there is at most one definition for every interface name I and that the fields and methods defined within an interface are all distinct). Interfaces are defined as follows.

Definition 8 *(Interface Definition Language).* Interface declarations ID are given by the language

$$\begin{array}{rcl} ID & ::= & \epsilon \mid \texttt{interface}\ \ I_1\ \ \texttt{:}\ \ I_2\ \ \{IF\ \ IM\}\ \ ID \\ IF & ::= & \epsilon \mid q : \mathsf{var}(B_s)\ \ IF \\ IM & ::= & \epsilon \mid f : \mathsf{proc}(\widetilde{B_s}) : s'\ \ IM \end{array}$$

■

In the declaration $\texttt{interface}\ \ I_1\ \ \texttt{:}\ \ I_2\ \ \{IF\ \ IM\}$, we have that I_1 is the name of the interface and I_2 is the name of its parent in the hierarchy. For the sake of simplicity, we assume that the members (i.e., fields and methods) common to both interfaces also appear in both the parent and the child, i.e., members are not automatically inherited. We also assume that all contracts implement only a single interface[6]. Furthermore, we assume the existence of an interface $I^\top$, corresponding

[6]We could also allow contracts to implement multiple interfaces; this would require the type of an address to be a *set* of interfaces, rather than just a single interface. However, we shall forego that here to avoid complicating the presentation.

to the basic implementation of a contract, which acts as the single root of the inheritance tree, corresponding to the following definition:

```
interface I⊤ : I⊤ {
  balance : var(int, s⊤)
  send    : proc() : s⊥
}
```

This definition may look strange, since it specifies a self-inheritance relation; however, we admit it since $I^\top$ is the root of the inheritance tree, and it alone will not inherit from anything. Moreover, the inclusion of a contract 'supertype' $I^\top_{s_\top}$, which results from assigning the maximum security level to the interface $I^\top$, is similar to what is done in the type system developed for FS by Crafa et al. in [16]. This is necessary to enable us to give a type to the 'magic variable' `sender` (see rule [T-ENV-M] in Figure 15 later on), which is available within the body of every method, since this variable can be bound to the address of any contract or account. In Figures 9 and 10 in the following section, we shall give a definition of a subtyping relation that will ensure that, for any valid interface definition I (containing at least `balance` and `send`) and any security level annotation s, it must hold that I_s is a subtype of $I^\top_{s_\top}$, thus always allowing us to type I_s up to $I^\top_{s_\top}$.

We shall also use a *typed* syntax of TinySol, where local variables are now declared as

$$\mathsf{var}(B_s)\ \ x\ \ \texttt{:=}\ \ e\ \ \texttt{in}\ \ S$$

where B is the type of the expression e and s is the security level declared for x. Likewise, we add annotated type names I_s to contract declarations:

$$\texttt{contract}\ \ X\ \ \texttt{:}\ \ I_s\ \ \{DF\ \ DM\}$$

where I is a declared interface name. Note that the security level is given on the *contract*, rather than on the interface declaration. This is intentional since multiple contracts may implement the same interface, but nevertheless be categorised into different security levels. Notice that the security level assigned to the contract (s for contract X above) regulates how that contract name is used as a value, but it also indirectly imposes constraints on its fields and methods. Indeed, in principle we could have a H-contract with L-fields, but none of them would be writable, because the typing rule for field assignments will require that the level of the contract must be lower than, or equal to, the level of the field (see rule [T-ASSF] in Figure 12). The same holds for methods, whose typing rule will require that the callee can be typed at level s, which in practice means it must be s or lower (see rule [T-CALL] in Figure 13). In contrast, we can have a L-contract with H-fields and H-methods.

Finally, type environments are used to record the interface hierarchy and other type assignments. In particular, we use the following type environments, as introduced in Definition 7:

- Σ : INames $\rightharpoonup$ INames records the immediate parent of an interface; thus, $\Sigma(I_1) = I_2$ whenever we have the declaration `interface` I_1 `:` I_2 $\{IF\ \ IM\}$.
- $\Gamma \in \mathcal{E} ::= ((\mathsf{FNames} \cup \{\texttt{balance}\} \cup \mathsf{MNames} \cup \mathsf{ANames}) \rightharpoonup \mathcal{T}) \cup (\mathsf{INames} \rightharpoonup \mathcal{E})$ records the statically declared definitions (for fields, methods, and contracts) and the types of interfaces. We store interfaces in Γ by letting interface names I return another type environment Γ_I, which records the signatures of fields and methods declared in interface I. Thus, if a contract with address X implements an interface I, we will have that $\Gamma(X) = I_s$, for some security level s, and $\Gamma(I) = \Gamma_I$, where Γ_I records the type annotations prescribed by IF and IM, provided that I is declared as `interface` I `:` I' $\{IF\ \ IM\}$.

— $\Delta : (\mathsf{VNames} \cup \{\mathtt{this}, \mathtt{sender}, \mathtt{value}\}) \rightharpoonup \mathcal{T}$ records the types of local variables. We record these in a separate environment since local variables are declared at runtime. Thus, this environment may change according to the runtime evolution of the program, whereas the other environments are static. This is in agreement with the operational semantics, where env_T remains the same and the domain of env_S is unaltered, whereas env_V usually changes.

For type environments of kind Γ, we further assume the following well-formedness condition:

Definition 9 *(Well-Formedness of* Γ*).* We say that a type environment Γ is *well-formed*, if the following hold:

— for each contract name X, if $X \in \mathrm{dom}(\Gamma)$, then there exists $I \in \mathsf{INames}$ such that $\Gamma(X) = I_s$ and $I \in \mathrm{dom}(\Gamma)$;
— for each interface name I, if $I \in \mathrm{dom}(\Gamma)$, then there exists $\Gamma_I \in \mathcal{E}$ such that $\Gamma(I) = \Gamma_I$. ■

Example 2. Let us now return to the code of Figure 1. Assume that `X` and `Y` implement interfaces I^X and I^Y, where

$$\mathtt{interface}\ I^X\ :\ I^\top\ \{\ \ldots\ \mathtt{called}\ :\ \mathsf{bool}_s\ \ \ldots\ \mathtt{deposit}\ :\ \ \mathsf{proc}(I^1_{s_1}) : s'_1\ \}$$
$$\mathtt{interface}\ I^Y\ :\ I^\top\ \{\ \ldots\ \mathtt{transfer}\ :\ \ \mathsf{proc}(I^2_{s_2}) : s'_2\ \}$$

and $I^1_{s_1}$ and $I^2_{s_2}$ are the types of the method arguments (to be assigned to variables `y` and `x`, respectively – that is, $\Delta(\mathtt{y}) = \mathsf{var}(I^1_{s_1})$ and $\Delta(\mathtt{x}) = \mathsf{var}(I^2_{s_2})$ during the execution of `deposit` and `transfer`). Then, $\{(I^X, I^\top), (I^Y, I^\top)\} \subseteq \Sigma$; moreover, $\Gamma(\mathtt{X}) = I^X_{s''_1}$ and $\Gamma(\mathtt{Y}) = I^Y_{s''_2}$, for some s''_1 and s''_2, and also $\{(\mathtt{called}, \mathsf{bool}_s), (\mathtt{deposit}, \mathsf{proc}(I^1_{s_1}) : s'_1)\} \subseteq \Gamma(I^X)$ and $\{(\mathtt{transfer}, \mathsf{proc}(I^2_{s_2}) : s'_2)\} \subseteq \Gamma(I^Y)$. ■

4.2 Subtyping

Σ gives a *static* representation of the subtype-supertype relationship for interfaces recorded in Γ; so, we need a way to decide whether a given Σ is *consistent* w.r.t. this relationship. This will be done through a set of rules for deciding whether a given interface I_1, declaring that it inherits from another interface I_2, indeed also is a *subtype* of I_2. The rules are given in Figure 8; the judgment is of the form $\Sigma; \Gamma \vdash \Sigma'$, which expresses that the inheritance tree recorded in Σ' is *consistent* with the actual declarations recorded in Σ and Γ.

The rules iterate through the inheritance environment, examining each entry in turn. Σ only records the inheritance hierarchy, while the actual structures of the interfaces are recorded in Γ; therefore, consistency of any subpart of Σ must be judged relative to Γ, as well as relative to the full Σ. The judgement can be better understood by thinking of Σ as representing the interfaces hierarchy bottom-up: the rightmost entry is for the root of the tree ($I^\top$); so, scanning Σ from left to right actually means to visit the hierarchy from the leaves to the root. The base case is the rule [Σ-TOP], since we assume that the interface $I^\top$ always exists and inherits from nothing besides itself; furthermore, the form of the conclusion ensures that this rule can only be applied to an inheritance tree consisting of the root node itself. The recursive case is the rule [Σ-REC], which examines the entry (I_1, I_2), for $I_1 \neq I_2$ (since no interface except $I^\top$ may inherit from itself). We also require that there must not exist another interface I' such that I' inherits from I_1 according to the remainder of the inheritance environment Σ'; this ensures that the environment indeed describes a *tree* and that [Σ-REC] is only applicable when I_1 is a *leaf node* of the current inheritance tree, which is then trimmed away as we recur in the premise. Since Σ represents the interface hierarchy bottom-up, requiring that the pair (I_1, I_2) is never followed by a pair (I', I_1) ensures that there is no cycle in the hierarchy: without this requirement, a cycle could be present, in the

$$[\textsc{sub-field}]\frac{\Sigma \vdash (B_1, s_1) <: (B_2, s_2)}{\Sigma \vdash \mathsf{var}(B_1, s_1) <: \mathsf{var}(B_2, s_2)}$$

$$[\textsc{sub-proc}]\frac{\Sigma \vdash (\widetilde{B_2, s_2}) <: (\widetilde{B_1, s_1})}{\Sigma \vdash \mathsf{proc}(\widetilde{B_1, s_1}) : s_1' <: \mathsf{proc}(\widetilde{B_2, s_2}) : s_2'} \ (s_2' \sqsubseteq s_1')$$

Fig. 9. Subtyping rules for interface members.

$$[\textsc{sub-refl}]\frac{}{\Sigma \vdash B <: B} \qquad [\textsc{sub-trans}]\frac{\Sigma \vdash B_1 <: B_2 \quad \Sigma \vdash B_2 <: B_3}{\Sigma \vdash B_1 <: B_3}$$

$$[\textsc{sub-name}]\frac{}{\Sigma \vdash I_1 <: I_2} \ (\Sigma(I_1) = I_2) \qquad [\textsc{sub-type}]\frac{\Sigma \vdash B_1 <: B_2}{\Sigma \vdash (B_1, s_1) <: (B_2, s_2)} \ (s_1 \sqsubseteq s_2)$$

Fig. 10. The subtyping relation for base types (B) and expression types (B_s).

case in which I_2 is (possibly indirectly) related to I' in Σ. Moreover, the remaining premises of [Σ-REC] ensure that all members of the supertype (I_2) also exist in the subtype (I_1) and, for each of them, that the subtyping judgment $\Sigma \vdash \Gamma_1(n) <: \Gamma_2(n)$ holds.

The rules defining the subtyping judgments for interface members are given in Figure 9, where $\Sigma \vdash (\widetilde{B, s}) <: (\widetilde{B', s'})$ requires that the two vectors have the same length, say h (≥ 0), and that the relation holds component-wise (i.e., $\Sigma \vdash (B_i, s_i) <: (B_i', s_i')$, for every $i \in \{1, \ldots, h\}$). Both rules invoke a subtyping judgment for types of the form $\Sigma \vdash (B_1, s_1) <: (B_2, s_2)$ in their premise: this, finally, is the proper subtyping relation, which is defined by the rules in Figure 10. These rules are very simple: they consist only of the reflexive and transitive rules, and of the rule [SUB-NAME], which reflects the interface declarations. For this reason, the subtyping relation is parametrised with Σ and in [SUB-NAME] we simply look up the supertype in Σ, rather than structurally comparing their respective definitions for I_1 and I_2 in Γ (notice that the comparison between the constituents of I_1 and I_2 within Γ is carried out in the premises of rule [Σ-REC], assuming that Σ is consistent with Γ). Hence, there is no circularity here: indeed, even though rule [Σ-REC] exploits $<:$ in its premises and this may involve some subtyping check on interfaces, we simply check that Σ contains an appropriate sequence of pairs (by means of rules [SUB-NAME] and [SUB-TRANS]).

Remark 3 *(On the subtyping of methods and fields).* Subtyping of interface members (fields and methods) is complicated by their variances, where reading is covariant and writing is contravariant. Methods are comparable to 'write-only' fields, since subtyping should not be invoked when we are typing the *declaration* of a method. Hence, the method type constructor is contravariant in its arguments, as can be seen in the rule [SUB-PROC]. However, due to this contravariance, we get the rather bizarre situation that an interface supertype must be less specific (i.e., have fewer members) as expected, but its methods must have *more* specific arguments than in the subtype. This hardly seems useful, but we nevertheless still allow it to give the rules a uniform shape.

With fields, the situation is more complicated. Usually in class-based/object-oriented languages, fields are *both* readable *and* writable. Hence, subtyping for fields should therefore be both covariant *and* contravariant, depending on whether we read from the field or write to it. In other words, it must be *invariant*, since allowing either form of variance would lead to an unsound subtyping relation, which, in the present setup, would mean that we could create an illegal information flow according to the security levels. However, as can be seen in the rule [SUB-FIELD], we actually allow subtyping of the field type constructor $\mathsf{var}(B_s)$ to be *covariant* in its argument. This is possible

because of a peculiar feature of TinySol: fields in this language are *not* writable outside of the contract in which they are declared. This is ensured directly in the syntax, since a field is only allowed to appear as the left-hand side of an assignment of the form `this.`p `:=` e. Thus, fields are actually *read-only* in all contexts in which subtyping may be invoked; therefore, we can allow covariant subtyping for fields. This is important since we would otherwise be unable to coerce an arbitrary contract up to $I^{\top}$, which is necessary to allow us to give a type to the 'magic variable' `sender`, which is available within every method body.

This is also the reason why we separate the subtyping rules for interface members (given in Figure 9 and used in rule [Σ-rec] for checking the consistency of Σ) from the definition of the proper subtyping relation (Figure 10), since only the latter may be used in the actual type judgments. In the following presentation, we shall therefore only allow subtyping to be used when judging the type of an expression e, which limits the usage of subtyping to the right-hand side of assignments and the arguments of a method call. This automatically ensures that the rules in Figure 9 cannot be invoked, since an expression cannot return a field name or a method name (i.e., something of type $\mathsf{var}(B_s)$ or $\mathsf{proc}(\widetilde{B_s}) : s'$); moreover, even though an expression can yield an address X of type I, we simply look it up in Σ by rule [sub-name]. ■

4.3 Type Judgments

We shall now present the rules for type judgments for the syntactic categories of TinySol(namely, values, expressions, and statements), for environments env_{TSV}, and lastly for contract declarations.

Types of Values. We can use a type environment Γ to assign types to values. To this end, we define the following partial function:

Definition 10 *(TypeOf).* The partial function $\textsc{TypeOf}_{\Gamma}(v)$ from values to types is defined as

$$\textsc{TypeOf}_{\Gamma}(v) = \begin{cases} (\mathsf{int}, s_{\bot}) & \text{if } v \in \mathbb{Z} \\ (\mathsf{bool}, s_{\bot}) & \text{if } v \in \mathbb{B} \\ \Gamma(v) & \text{if } v \in \mathsf{ANames} \text{ and } v \in \mathrm{dom}\,(\Gamma) \end{cases}$$

and is undefined, otherwise. ■

$\textsc{TypeOf}_{\Gamma}(\cdot)$ assigns basic types to values in an obvious way, based on the set to which they belong. However, since the type of a value has both a *data* aspect and a *security* aspect, we also need to assign a security level to values. We choose the lowest possible level ($s_{\bot}$) for the 'pure' data values (integers and booleans), since they have no inherent security level attached to them. Hence, it is always safe to give them the lowest possible level and then coerce it up to a higher level via subtyping, when we write the value into a variable/field (see rules [T-Assv]/[T-Assf] later on). Thence, the level will be determined by the level of the variable/field. In contrast, the interface and the security level of an address name is provided directly by the specified type environment Γ.

4.3.1 Safety Requirement for Operations. We have left the syntactic category of operations op undefined and simply assumed that they can all be evaluated using some total deterministic relation $\rightarrow_{\mathrm{op}}$, such that $\mathrm{op}(\tilde{v}) \rightarrow_{\mathrm{op}} v$. Furthermore, we assume that, for each operation, we can determine its signature, written $\vdash \mathrm{op} : \widetilde{B} \rightarrow B$. To tie these two together, we need to make one further assumption, namely that, if the signature of an operation is $\widetilde{B} \rightarrow B$ and the actual parameters $\tilde{v}$ are of basic types $\widetilde{B}$ (with $|\tilde{v}| = |\widetilde{B}|$), then the operation can actually be performed and the resulting value v is of type B. As neither the operations, their semantics, nor their typing rules are specified, all we can do w.r.t. safety is to require that this must hold. Formally, we can specify the requirement as follows.

$$[\text{T-VAL}]\frac{}{\Sigma;\Gamma;\Delta \vdash v : B_s}\ (\textsc{TypeOf}_\Gamma(v) = B_s)$$

$$[\text{T-VAR}]\frac{}{\Sigma;\Gamma;\Delta \vdash k : B_s}\ (\Delta(k) = \mathsf{var}(B_s))$$

$$[\text{T-FIELD}]\frac{\Sigma;\Gamma;\Delta \vdash e : I_s}{\Sigma;\Gamma;\Delta \vdash e.q : B_s}\ (\Gamma(I)(q) = \mathsf{var}(B_s))$$

$$[\text{T-OP}]\frac{\vdash \mathrm{op} : \widetilde{B} \to B \quad \forall i\,.\,\Sigma;\Gamma;\Delta \vdash e_i : (B_i, s)}{\Sigma;\Gamma;\Delta \vdash \mathrm{op}(\tilde{e}) : B_s}\ \left(|\tilde{e}| = |\widetilde{B}|\right)$$

$$[\text{T-ESUB}]\frac{\Sigma;\Gamma;\Delta \vdash e : (B_1, s_1) \quad \Sigma \vdash (B_1, s_1) <: (B_2, s_2)}{\Sigma;\Gamma;\Delta \vdash e : (B_2, s_2)}$$

Fig. 11. Typing rules for expressions.

Definition 11 *(Safety Requirement for Operations).* We say that Γ satisfies the safety requirement for operations w.r.t. $\to_{\mathrm{op}}$ if, for all operations op and for any argument list $\tilde{v} = v_1, \ldots, v_h$, it holds that

—$\vdash \mathrm{op} : \widetilde{B} \to B$, with $\widetilde{B} = B_1, \ldots, B_h$, and
—$\forall i \in \{1, \ldots, h\}\,.\,\textsc{TypeOf}_\Gamma(v_i) = (B_i, s_i)$, for some s_i,

imply that

—for all $X \in \mathsf{ANames}$ such that $X \in \tilde{v}$, it holds that $X \in \mathrm{dom}\,(\Gamma)$, and
—$\textsc{TypeOf}_\Gamma(v) = (B, s_\bot)$, where $\mathrm{op}(\tilde{v}) \to_{\mathrm{op}} v$. ■

Note that, with this definition, we allow addresses to appear as arguments for operations. However, since we assume that an operation can never yield an address, we also know that the resulting security level will be $s_\bot$, and that the resulting base type B will be one of int or bool. Moreover, operations are only concerned with the actual data and not with their security levels, since all arguments must be actual values, i.e., not variables. Therefore, in Definition 11 we disregard the security levels; the security level will be checked in the typing rule for operations given in the following section. This simply ensures that we can give syntax-directed typing rules for expressions.

4.3.2 Typing Expressions. We start with the typing rules for expressions, given in Figure 11. Here, judgments are of the form $\Sigma;\Gamma;\Delta \vdash e : B_s$, indicating that all reads needed to evaluate e are from containers of security level s or lower and that the resulting value is of type B, modulo the type environments Σ, Γ and Δ. There are a few things to note:

—In rule [T-VAL], the type of a value v is determined by the $\textsc{TypeOf}_\Gamma(\cdot)$ function, which, in the case of integers and booleans, will set the security level to $s_\bot$, which is the lowest level. This is a consequence of the fact that there is no relationship between the datatype of a value and its security level. The level can then be raised safely as required by subtyping, and therefore the actual security level will be determined by the type of the variable (resp. field) to which the value is assigned.
—The rules [T-VAR] and [T-FIELD] simply unwrap the type of the contained value from the type of the container. Remember from Section 2.2.2 that k ranges over all variable names, including

$$[\text{T-SKIP}]\frac{}{\Sigma;\Gamma;\Delta \vdash \texttt{skip} : \mathsf{cmd}(s)}$$

$$[\text{T-THROW}]\frac{}{\Sigma;\Gamma;\Delta \vdash \texttt{throw} : \mathsf{cmd}(s)}$$

$$[\text{T-DECV}]\frac{\Sigma;\Gamma;\Delta \vdash e : B_{s'} \quad \Sigma;\Gamma;\Delta, x : \mathsf{var}(B_{s'}) \vdash S : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash \mathsf{var}(B_{s'})\ \ x\ \ \texttt{:=}\ \ e\ \ \texttt{in}\ \ S : \mathsf{cmd}(s)}$$

$$[\text{T-WHILE}]\frac{\Sigma;\Gamma;\Delta \vdash e : \mathsf{bool}_s \quad \Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash \texttt{while}\ \ e\ \ \texttt{do}\ \ S : \mathsf{cmd}(s)}$$

$$[\text{T-IF}]\frac{\Sigma;\Gamma;\Delta \vdash S_{\mathsf{T}} : \mathsf{cmd}(s) \quad \Sigma;\Gamma;\Delta \vdash e : \mathsf{bool}_s \quad \Sigma;\Gamma;\Delta \vdash S_{\mathsf{F}} : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash \texttt{if}\ \ e\ \ \texttt{then}\ \ S_{\mathsf{T}}\ \ \texttt{else}\ \ S_{\mathsf{F}} : \mathsf{cmd}(s)}$$

$$[\text{T-SEQ}]\frac{\Sigma;\Gamma;\Delta \vdash S_1 : \mathsf{cmd}(s) \quad \Sigma;\Gamma;\Delta \vdash S_2 : \mathsf{cmd}(s)}{\Sigma;\Gamma;\Delta \vdash S_1; S_2 : \mathsf{cmd}(s)}$$

$$[\text{T-ASSV}]\frac{\Sigma;\Gamma;\Delta \vdash e : B_s}{\Sigma;\Gamma;\Delta \vdash x\ \ \texttt{:=}\ \ e : \mathsf{cmd}(s)}$$
where:
$$\Delta(x) = \mathsf{var}(B_s)$$

$$[\text{T-ASSF}]\frac{\Sigma;\Gamma;\Delta \vdash e : B_s}{\Sigma;\Gamma;\Delta \vdash \texttt{this}.p\ \ \texttt{:=}\ \ e : \mathsf{cmd}(s)}\ \ (s' \sqsubseteq s)$$
where:
$$\Delta(\texttt{this}) = I_{s'} \text{ and } \Gamma(I)(p) = \mathsf{var}(B_s)$$

$$[\text{T-SSUB}]\frac{\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s_1)}{\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s_2)}\ \ (s_2 \sqsubseteq s_1)$$

Fig. 12. Typing rules for statements – 1.

the 'magic variables' `this`, `sender` and `value`, and that q ranges over all field names, including `balance`.

—Finally, in rule [T-OP], we require that all arguments and the return value must be typable to the same security level s. Recall that we assume that no operation has an address as return type. Operations may have addresses as some of their *arguments* (e.g., for equality testing), but the return type must be one of the other base types, which can be given a security level. Thus, in rule [T-OP], we also need to extract the security level s from the types of the argument expressions.

4.3.3 Typing Statements. Next, we consider the typing rules for statements; they are given in Figures 12 and 13. Here, judgments are of the form $\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s)$, indicating that all variables *written to* within S are of level s or *higher*, under Σ, Γ and Δ. To ensure this property, the basic rules are those for assignments (to either variables or fields) and for subtyping: the former ones impose that the level of the command is the level of the assigned variable/field; the latter one states that the level can be arbitrarily downgraded. A local variable declaration has no effect on

$$[\text{T-CALL}]\ \frac{\Sigma;\Gamma;\Delta \vdash e_1 : (I^Y, s) \quad \Sigma;\Gamma;\Delta \vdash e_2 : \mathsf{int}_s \quad \Sigma;\Gamma;\Delta \vdash \tilde{e} : \widetilde{B_{s'}}}{\Sigma;\Gamma;\Delta \vdash e_1.f(\tilde{e})\$e_2 : \mathsf{cmd}(s)} \begin{pmatrix} s_1 \sqsubseteq s \\ s \sqsubseteq s_2, s_3 \end{pmatrix}$$

where:

$$\begin{aligned} \Delta(\mathtt{this}) &= \mathsf{var}(I^X, s_1) \\ \Gamma(I^X)(\mathtt{balance}) &= \mathsf{var}(\mathsf{int}, s_2) \\ \Gamma(I^Y)(\mathtt{balance}) &= \mathsf{var}(\mathsf{int}, s_3) \\ \Gamma(I^Y)(f) &= \mathsf{proc}(\widetilde{B_{s'}}) : s \end{aligned}$$

$$[\text{T-DCALL}]\ \frac{\Sigma;\Gamma;\Delta \vdash e : (I^Y, s) \quad \Sigma \vdash (I^X, s_1) <: (I^Y, s) \quad \Sigma;\Gamma;\Delta \vdash \tilde{e} : \widetilde{B_{s'}}}{\Sigma;\Gamma;\Delta \vdash \mathtt{dcall}\ \ e.f(\tilde{e}) : \mathsf{cmd}(s)}$$

where:

$$\begin{aligned} &\Delta(\mathtt{this}) = \mathsf{var}(I^X, s_1) \\ &\Gamma(I^Y)(f) = \mathsf{proc}(\widetilde{B_{s'}}) : s \\ &\forall q \in (\mathsf{dom}(\Gamma(I^Y)) \cap (\mathsf{FNames} \cup \{\mathtt{balance}\})).\ \Gamma(I^Y)(q) = \Gamma(I^X)(q) \end{aligned}$$

Fig. 13. Typing rules for statements – 2.

the security level of the command: however, the compliance with the current level will be checked whenever assigning that variable, by requiring that the level of the assigned variable is at least the s in $\mathsf{cmd}(s)$. Commands `skip` and `throw` do not write any variable, so they can be assigned any security level. Thanks to rule [T-SSUB], a sequential composition can take any security level that is smaller than, or equal to, the greatest lower bound of the security levels of its components. For guarded commands (i.e., `if` and `while`), we impose that the subcommand(s) can be typed at the level of the guard; this is standard in the setting of noninterference since the evaluation of the guard influences the continuation of the computation.

In rules [T-ASSF], [T-CALL] and [T-DCALL], we are reading an address (respectively, from the magic variable `this` and expression e_1) and then we use that address to access a member (either a field or a method name). So, following the principle that nothing must depend on information that is of a level higher than or incomparable to itself, we must have that the members must be of a security level that is higher than or equal to the level of the address through which they are accessed. This justifies the side conditions $s' \sqsubseteq s$ in [T-ASSF] and $s_1 \sqsubseteq s$ in [T-CALL]; notice that the latter condition is also implicit in [T-DCALL], because of the premise $\Sigma \vdash (I^X, s_1) <: (I^Y, s)$.

There are other subtleties in the rules for method calls (i.e., [T-CALL] and [T-DCALL]), so we discuss them in detail. According to the operational rule [CALL], every call includes an implicit read and write of the `balance` field of the caller contract, since the call will only be performed if the value of e_2 is less than or equal to the value of `balance` (to ensure that the subtraction will not yield a negative number). There is thus an implicit flow from `this.balance` to the body S of the method call, similar to the case for the guard expression e in an if- or in a while-statement.

Furthermore, there is an implicit write to the `balance` field of the callee contract, and thus a flow of information from one field to the other. This might initially seem like it would require both the caller and the callee to have the same security level for their field `balance`. However, the levels *can* differ, since by subtyping we can coerce one up to match the level of the other. For this reason, we have the side condition $s \sqsubseteq s_2, s_3$, where s_2 and s_3 are the levels of the `balance` fields of the caller and callee, respectively. This enables calls from a lower security level into a higher security level, but not the other way around. The reason behind this condition is as follows. Suppose that the *actual* level of e_2 was some level s'. According to the semantics (Figure 6), every method call

implicitly performs a write to the balance fields of both the caller and the callee, corresponding to the following two lines of code (where Y is the address of the callee):

```
this.balance := this.balance - e2;
Y.balance := Y.balance + e2
```

Typing these statements as in [T-ASSF] would give us that $s' \sqsubseteq s_2$ and $s' \sqsubseteq s_3$; hence $s' \sqsubseteq s_2, s_3$. In both cases, the level of the left-hand side must of course be above-or-equal to the level of the right-hand side, and since `this.balance` (resp. Y`.balance`) appears on both sides, we have that s' must be less-than or equal-to both, otherwise the level of the right-hand side would be above the level of the left-hand side. Finally, since both `balance` fields are being modified in the method body, albeit implicitly, it must also be the case that $s \sqsubseteq s_2, s_3$, where s is the level of the method body. We then simplify this by requiring that e_2 can also be evaluated at the level s, which it can by subtyping if the condition $s' \sqsubseteq s$ holds. Notice that the *value* of e_2 can influence the behaviour of the program, namely if its runtime evaluation yields a value that is strictly greater than the current balance of the caller. Thus, there could, in principle, be a flow of information from the `balance` field of the caller, *if* it were possible to observe a call that fails due to insufficient funds in the caller contract. However, the condition $s_2 \sqsubseteq s$ is *not* needed in rule [T-CALL], since the runtime semantics simply blocks the execution of a method call with a transferred value that is greater than the current balance. We can therefore ignore these cases here, since all our definitions and results assume that the programs correctly terminate. However, if exceptions and roll-back were added to the language, as in e.g., [16], this condition would be necessary as well.

The case for delegate calls is only seemingly simpler, since there is no currency transfer associated with the call. However, there are two important points to note in rule [T-DCALL]. First, we have the subtyping judgment $\Sigma \vdash (I^X, s_1) <: (I^Y, s)$ to require that the interface type of the caller contract is a subtype of the contract that contains the definition of f. This is necessary because the body of f might access fields or call methods via `this`; but when the body is executed in the context of the *caller*, this would lead to a runtime error if the calling contract does not contain fields or methods with the same names (and of the same types). This situation is prevented by requiring the caller contract to be a subtype (i.e., a specialisation) of the callee contract, thus ensuring that the caller contract will contain *at least* the same methods and fields as the callee contract. Note that this implies that the *caller contract* also must contain a method with the same name f and with the same signature (or a subtype thereof). We could remove this limitation by introducing a more complicated typing rule, but we shall forego that here, since it is consistent with the purpose of `dcall`, i.e., code reuse. Second, requiring the aforementioned sybtyping judgement is not enough for fields: indeed, we must impose that all fields of I^Y have the *very same type* as in I^X. Without this requirement, it is possible to have an insecure flow, as the code of Figure 14 shows. The body of `X.f` is well-typed: it has a H security level, since it writes a H value into a H field. Also `Y` is well-typed; moreover, `I2` <: `I1` because the types of the common members are either the same, or have a lower security level in case of the fields. However, executing `Y.g()` leads to `this.a := X.b` in a context where `this` is bound to `Y`; so, we get `Y.a := X.b`, which is a flow from a H field to a L one! This is possible, because the subtyping for fields is co-variant (when checking the consistency of Σ), which is fine without delegate calls; indeed, fields are only writable within the contract in which they are declared. However, with delegate calls, we can pull in code from a different context and 'capture' fields, which can now suddenly have a different security level. To sum up, with delegate calls, subtyping of fields (between caller and callee) must be *contra-variant*, since any field in the callee must also be present in the caller (in principle, it can occur on the left-hand side of an assignment); but since we also in general require them to be subtypes (i.e., *co-variant*), subtyping for the common fields must therefore in practice be *invariant*. This is ensured by the side

```
contract X : I1_H {
  ...
  field a := 0;
  field b := 2;
  f() { this.a := X.b }
  ...
}
```

```
interface I1 : I^⊤ {
  ...
  a : var(int, H)
  b : var(int, H)
  f : proc():H
  ...
}
```

```
contract Y : I2_H {
  ...
  field a := 0;
  field b := 0;
  f() { skip }
  g() { dcall X.f() }
  ...
}
```

```
interface I2 : I1 {
  ...
  a : var(int, L)
  b : var(int, L)
  f : proc():H
  g : proc():H
  ...
}
```

Fig. 14. A code with delegate calls and the potential risk of information flow from H to L.

condition in rule [T-DCALL], which states that, for each field q common to the caller and the callee, q must have the same type in both interfaces.

Finally, note that in both [T-CALL] and [T-DCALL], in the list of types $\widetilde{B_{s'}}$, the levels $\widetilde{s'}$, which we shall expand as $s'_1, \ldots, s'_h$, do not have to be related to any of the other levels.

4.3.4 Typing Environments. After the initial reduction step, all declarations are stored in the two environments env_{ST}, and further reductions also use the variable environment env_V for local variable declarations. Hence, we also need to be able to conclude *agreement* between these environments and the type environments Σ, Γ and Δ; this is given in Figure 15. Note that the check here only ensures that every declared contract member has a type; the converse check (i.e., that every declared type in an interface also has an implementation) should also be performed. However, we shall omit this in the present treatment and simply assume that it holds.

For all environments, the type judgments are relative to a Σ, Γ and Δ, to make them uniform. However, the Δ component is actually needed only for typing env_V. In particular, [T-ENV-F] invokes [T-VAL] to conclude a type for the value v stored in a field q, but this rule does not rely on Δ since v is a value and so does not contain any variable to be evaluated. In contrast, in [T-ENV-M] we build a new Δ' from the signature of the method to be able to type the body S in the premise. Moreover, when typing a method declaration, we do not statically know which contracts will invoke it; hence, we have to assume that `sender` may be any contract (thus, the typing assignment $\mathsf{var}(I^\top, s_\top)$ that, by subtyping, encompasses all contracts). Finally, for fields and methods, we must know the address of the contract containing them, to retrieve the associated typing information from Γ (this is the role of the X labelling $\vdash$ in the rules on the left-hand side of Figure 15). Notice however that we do not need to compare the security level of a field/method with the level of the contract that contains it since this task is carried out when modifying the field or invoking the method, respectively (see rules [T-ASSF] and [T-CALL]).

4.3.5 Typing Declarations. Finally, in Figure 16, we give the type judgments for contract declarations, that closely follow the rules for environment agreement from Figure 15 (compare the side conditions of rules [T-ENV-M] and [T-ENV-F] with those of rules [T-DEC-M] and [T-DEC-F]). This similarity is not surprising, since the declarations DM and DF and environments env_{TS} are two equivalent representations of the given code. Judgments are of the form $\Sigma; \Gamma; \Delta \vdash DC$, stating

$$[\text{T-ENV-}\emptyset_1]\frac{}{\Sigma;\Gamma;\Delta \vdash_X \mathsf{env}_J^{\emptyset}} \ (J \in \{F, M\})$$

$$[\text{T-ENV-}\emptyset_2]\frac{}{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_J^{\emptyset}} \ (J \in \{T, S, V\})$$

$$[\text{T-ENV-F}]\frac{\Sigma;\Gamma;\Delta \vdash v : B_s \quad \Sigma;\Gamma;\Delta \vdash_X \mathsf{env}_F}{\Sigma;\Gamma;\Delta \vdash_X \{(q, v)\} \cup \mathsf{env}_F}$$
where:
$\Gamma(X) = (I, s')$ and $\Gamma(I)(q) = \mathsf{var}(B_s)$

$$[\text{T-ENV-T}]\frac{\Sigma;\Gamma;\Delta \vdash_X \mathsf{env}_M \quad \Sigma;\Gamma;\Delta \vdash \mathsf{env}_T}{\Sigma;\Gamma;\Delta \vdash \{(X, \mathsf{env}_M)\} \cup \mathsf{env}_T}$$

$$[\text{T-ENV-M}]\frac{\Sigma;\Gamma;\Delta' \vdash S : \mathsf{cmd}(s) \quad \Sigma;\Gamma;\Delta \vdash_X \mathsf{env}_M}{\Sigma;\Gamma;\Delta \vdash_X \{(f, (x_1, \dots, x_h, S))\} \cup \mathsf{env}_M}$$
where:
$\Gamma(X) = (I, s_1)$
$\Gamma(I)(\mathtt{balance}) = \mathsf{var}(\mathsf{int}, s_2)$
$\Gamma(I)(f) = \mathsf{proc}((B_1, s_1'), \dots, (B_h, s_h')) : s$
$\Delta' = \mathtt{this} : \mathsf{var}(I, s_1),$
$\mathtt{value} : \mathsf{var}(\mathsf{int}, s_2),$
$\mathtt{sender} : \mathsf{var}(I^{\top}, s_{\top}),$
$x_1 : \mathsf{var}(B_1, s_1'),$
$\dots$
$x_h : \mathsf{var}(B_h, s_h')$

$$[\text{T-ENV-S}]\frac{\Sigma;\Gamma;\Delta \vdash_X \mathsf{env}_F \quad \Sigma;\Gamma;\Delta \vdash \mathsf{env}_S}{\Sigma;\Gamma;\Delta \vdash \{(X, \mathsf{env}_F)\} \cup \mathsf{env}_S}$$

$$[\text{T-ENV-V}]\frac{\Sigma;\Gamma;\Delta \vdash v : B_s \quad \Sigma;\Gamma;\Delta \vdash \mathsf{env}_V}{\Sigma;\Gamma;\Delta \vdash \{(k, v)\} \cup \mathsf{env}_V}$$
where:
$\Delta(k) = \mathsf{var}(B_s)$

$$[\text{T-ENV-SV}]\frac{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_S \quad \Sigma;\Gamma;\Delta \vdash \mathsf{env}_V}{\Sigma;\Gamma;\Delta \vdash \mathsf{env}_{SV}}$$

Fig. 15. Typing rules for the method-, state- and variable environments.

$$[\text{T-DEC-}\emptyset_1]\frac{}{\Sigma;\Gamma;\Delta \vdash \epsilon}$$

$$[\text{T-DEC-}\emptyset_2]\frac{}{\Sigma;\Gamma;\Delta \vdash_X \epsilon}$$

$$[\text{T-DEC-F}]\frac{\Sigma;\Gamma;\Delta \vdash v : B_s \quad \Sigma;\Gamma;\Delta \vdash_X DF}{\Sigma;\Gamma;\Delta \vdash_X \mathtt{field}\ q\ \mathtt{:=}\ v;\ DF}$$
where:
$\Gamma(X) = I_{s'}$ and $\Gamma(I)(q) = \mathsf{var}(B_s)$

$$[\text{T-DEC-M}]\frac{\Sigma;\Gamma;\Delta' \vdash S : \mathsf{cmd}(s) \quad \Sigma;\Gamma;\Delta \vdash_X DM}{\Sigma;\Gamma;\Delta \vdash_X f(x_1, \dots, x_h)\ \{S\}\ DM}$$
where:
$\Gamma(X) = (I, s_1)$
$\Gamma(I)(\mathtt{balance}) = \mathsf{var}(\mathsf{int}, s_2)$
$\Gamma(I)(f) = \mathsf{proc}((B_1, s_1'), \dots, (B_h, s_h')) : s$
$\Delta' = \mathtt{this} : \mathsf{var}(I, s_1),$
$\mathtt{value} : \mathsf{var}(\mathsf{int}, s_2),$
$\mathtt{sender} : \mathsf{var}(I^{\top}, s_{\top}),$
$x_1 : \mathsf{var}(B_1, s_1'),$
$\dots$
$x_h : \mathsf{var}(B_h, s_h')$

$$[\text{T-DEC-C}]\frac{\Sigma;\Gamma;\Delta \vdash_X DF \quad \Sigma;\Gamma;\Delta \vdash_X DM \quad \Sigma;\Gamma;\Delta \vdash DC}{\Sigma;\Gamma;\Delta \vdash \mathtt{contract}\ X\ :\ I_s\ \{DF\ DM\}\ DC} \ (\Gamma(X) = I_s)$$

Fig. 16. Type rules for declarations.

that the declarations DC are *well-typed* w.r.t. the environments $\Sigma;\Gamma;\Delta$ (here Δ is never used, so it can be easily considered as $\emptyset$; we keep it general for the sake of uniformity): this holds if the declarations are consistent with the type information recorded in the environments, i.e., every field and method must have a type (again, for this reason we must record the address of the contract that contains such fields/methods), and the body of each method must be typable according to the assumptions of the type. Note that also here we omit the rules for ensuring that all declared types in an interface have an implementation in any contract claiming to implement that interface.

4.4 Safety and Soundness

As is the case for the type system proposed in [53], our type system does not have a now-safety predicate in the usual sense, since (invariant) safety in simple type systems is a 1-property, whereas noninterference is a hyper-property (specifically, a 2-property) [15]. Instead, the meaning of 'safety' is expressed directly in the meaning of the types:

—If an expression e has type B_s, then all variables *read* in the evaluation of e are of level s or *lower*, i.e., no read-up.
—If a statement S has type $\mathsf{cmd}(s)$, then all variables *written* in the execution of S are of level s or *higher*, i.e., no write-down.

Intuitively, the meaning of these two types together implies that information from higher-level variables cannot flow into lower-level variables. For a statement such as $x := e$ to be well-typed, it must therefore be the case that, if $\Sigma;\Gamma;\Delta \vdash x : \mathsf{var}(B_s)$ and $\Sigma;\Gamma;\Delta \vdash e : B_{s'}$, then $s' \sqsubseteq s$. Since s' can be coerced up to s through subtyping to match the level of the variable, the statement itself can then be typed as $\mathsf{cmd}(s)$. We shall prove that our type system indeed ensures these properties in Theorems 1–4 later.

Before proceeding, we generalise the notion of low-equivalence $=_L$ introduced in Section 3 to express that two *states*, i.e., two collections of variable and field environments env_{SV}, are equal up to a certain security level s. This relation, written $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$, is given by the rules in Figure 17. Note in particular that the definition implies that env^1_{SV} and env^2_{SV} must have the same domain, and this carries over to the inner environments env_F inside env_S. Given our annotation of security levels on interfaces as well, we also extend the $=_s$ relation to method tables env_T, and finally to the combined representation of state and code, i.e., env_{ST}. The following result follows directly from the definition of the s-parameterised equivalence relation and can be shown by a straightforward induction on the rules defining $=_s$:

Lemma 1 (Restriction). *If* $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$ *and* $s' \sqsubseteq s$, *then* $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_{s'} \mathsf{env}^2_{SV}$.

Next, we need the standard lemmas for strengthening and weakening of the variable environment env_V. However, as our types have both a data type aspect and a security aspect, this will also be reflected in the lemmas, which features two different forms of preservation; namely of well-typedness and of s-equality. Hence, the individual statements in these lemmas are shown by two separate inductions; i.e., on the typing rules and on the rules defining $=_s$, respectively.

Lemma 2 (Strengthening).

—*If* $\Sigma;\Gamma;\Delta, k : \mathsf{var}(B_{s'}) \vdash \{(k, v)\} \cup \mathsf{env}_V$ *then also* $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V$.
—*If* $\Gamma;\Delta, k : \mathsf{var}(B_{s'}) \vdash \{(k, v_1)\} \cup \mathsf{env}^1_V =_s \{(k, v_2)\} \cup \mathsf{env}^2_V$ *then also* $\Gamma;\Delta \vdash \mathsf{env}^1_V =_s \mathsf{env}^2_V$.

Lemma 3 (Weakening).

—*If* $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_V$ *and* $k \notin \mathrm{dom}(\mathsf{env}_V) \cup \mathrm{dom}(\Delta)$, *then, for any* B, s' *and* v *of type* B, *it holds that* $\Sigma;\Gamma;\Delta, k : \mathsf{var}(B_{s'}) \vdash \{(k, v)\} \cup \mathsf{env}_V$.
—*If* $\Gamma;\Delta \vdash \mathsf{env}^1_V =_s \mathsf{env}^2_V$ *and* $k \notin \mathrm{dom}(\mathsf{env}^1_V) \cup \mathrm{dom}(\mathsf{env}^2_V) \cup \mathrm{dom}(\Delta)$, *then, for any* B, s' *and* v *of type* B, *it holds that* $\Gamma;\Delta, k : \mathsf{var}(B_{s'}) \vdash \{(k, v)\} \cup \mathsf{env}^1_V =_s \{(k, v)\} \cup \mathsf{env}^2_V$.

Both lemmas can be extended directly to $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$ w.r.t. the security aspect; the data type aspect is not relevant to extend, since env_S does not change. Next, we also need another standard lemma, which shows that well-typedness is preserved by substitution in the two environments env_V and env_S.

Lemma 4 (Substitution).

$$[\text{EQ-ENV}_V^{\emptyset}]\frac{}{\Delta \vdash \mathsf{env}_V^{\emptyset} =_s \mathsf{env}_V^{\emptyset}}$$

$$[\text{EQ-ENV}_V]\frac{\Delta \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2}{\Delta \vdash \{(k, v_1)\} \cup \mathsf{env}_V^1 =_s \{(k, v_2)\} \cup \mathsf{env}_V^2} \begin{pmatrix} \Delta(k) = \mathsf{var}(B_{s'}) \\ s' \sqsubseteq s \implies v_1 = v_2 \end{pmatrix}$$

$$[\text{EQ-ENV}_F^{\emptyset}]\frac{}{\Gamma \vdash_I \mathsf{env}_F^{\emptyset} =_s \mathsf{env}_F^{\emptyset}}$$

$$[\text{EQ-ENV}_F]\frac{\Gamma \vdash_I \mathsf{env}_F^1 =_s \mathsf{env}_F^2}{\Gamma \vdash_I \{(q, v_1)\} \cup \mathsf{env}_F^1 =_s \{(q, v_2)\} \cup \mathsf{env}_F^2} \begin{pmatrix} \Gamma(I)(q) = \mathsf{var}(B_{s'}) \\ s' \sqsubseteq s \implies v_1 = v_2 \end{pmatrix}$$

$$[\text{EQ-ENV}_S^{\emptyset}]\frac{}{\Gamma \vdash \mathsf{env}_S^{\emptyset} =_s \mathsf{env}_S^{\emptyset}}$$

$$[\text{EQ-ENV}_S]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \qquad s' \sqsubseteq s \implies \Gamma \vdash_I \mathsf{env}_F^1 =_s \mathsf{env}_F^2}{\Gamma \vdash \{(X, \mathsf{env}_F^1)\} \cup \mathsf{env}_S^1 =_s \{(X, \mathsf{env}_F^2)\} \cup \mathsf{env}_S^2} \ (\Gamma(X) = I_{s'})$$

$$[\text{EQ-ENV}_T]\frac{\Gamma \vdash \mathsf{env}_T^1 =_s \mathsf{env}_T^2}{\Gamma \vdash \{(X, \mathsf{env}_M^1)\} \cup \mathsf{env}_T^1 =_s \{(X, \mathsf{env}_M^2)\} \cup \mathsf{env}_T^2} \begin{pmatrix} \Gamma(X) = I_{s'} \\ s' \sqsubseteq s \implies \mathsf{env}_M^1 = \mathsf{env}_M^2 \end{pmatrix}$$

$$[\text{EQ-ENV}_{SV}]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Delta \vdash \mathsf{env}_V^1 =_s \mathsf{env}_V^2}{\Gamma; \Delta \vdash \mathsf{env}_{SV}^1 =_s \mathsf{env}_{SV}^2}$$

$$[\text{EQ-ENV}_{ST}]\frac{\Gamma \vdash \mathsf{env}_S^1 =_s \mathsf{env}_S^2 \quad \Gamma \vdash \mathsf{env}_T^1 =_s \mathsf{env}_T^2}{\Gamma \vdash \mathsf{env}_{ST}^1 =_s \mathsf{env}_{ST}^2}$$

Fig. 17. Rules for the s-parameterised equivalence relation.

- *If* $\Sigma; \Gamma; \Delta, k : \mathsf{var}(B_s) \vdash \{(k, v_1)\} \cup \mathsf{env}_V$ *and* $\Sigma; \Gamma; \Delta, k : \mathsf{var}(B_s) \vdash v_2 : B_s$ *then also* $\Sigma; \Gamma; \Delta, k : \mathsf{var}(B_s) \vdash \{(k, v_2)\} \cup \mathsf{env}_V$.
- *Let* $\Gamma(X) = (I, s')$ *and* $\Gamma(I)(q) = \mathsf{var}(B_s)$. *If* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[q \mapsto v_1]]$ *and* $\Sigma; \Gamma; \Delta \vdash v_2 : B_s$, *then also* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S[X \mapsto \mathsf{env}_F[q \mapsto v_2]]$.

With this, we can now state the first of our main theorems, which says that expressions evaluate to a value of the correct type. The proof is by induction on the derivation of $\Sigma; \Gamma; \Delta \vdash e : B_s$ and is in Appendix A.

Theorem 1 (Data type safety for expressions). *If* $\Sigma; \Gamma; \Delta \vdash e : B_s$ *and* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$ *and* $\mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} v$, *then also* $\Sigma; \Gamma; \Delta \vdash v : B_s$.

Our next theorem assures us that the type of an expression is also in accordance with the intended meaning, namely: if $\Sigma; \Gamma; \Delta \vdash e : B_s$, then every variable or field read to evaluate e will be of level s or lower (i.e., no read-down). We express this requirement by considering two different states, env_{SV}^1 and env_{SV}^2, which must agree on all values of level s and lower; then, evaluating e w.r.t. either of these states should yield the same result. The proof is by induction on the derivation of $\Sigma; \Gamma; \Delta \vdash e : B_s$ and is in Appendix B.

Theorem 2 (Security type safety for expressions). *Assume that* $\Sigma;\Gamma;\Delta \vdash e : B_s$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}^1_{SV}$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}^2_{SV}$, *and* $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$. *Then,* $\mathsf{env}^1_{SV} \vdash e \to_{\mathrm{e}} v$ *and* $\mathsf{env}^2_{SV} \vdash e \to_{\mathrm{e}} v$.

Next, we need two special lemmas to handle the addition of delegate calls to the language. As previously mentioned, the body S of a delegate call is executed in the context of the *caller*, rather than the callee, but from the rules [T-ENV-M] and [T-DEC-M], we only know that the body is well-typed w.r.t. a type environment Δ in which the special variable `this` has the type of the *callee* contract; i.e., the contract containing the declaration. Furthermore, the magic variable `value` is also not rebound; although its data type (int) is always the same, its security level may be different from the context in which the body was typed. The following lemmas state that, if the caller contract is a subtype of the callee contract and we furthermore impose the restriction that subtyping of the fields is invariant, then these differences do not matter. They are shown by induction on the typing rules for expressions (Figure 11) resp. statements (Figures 12 and 13). The proofs are given in Appendix C and D.

Lemma 5. *Let* $\Sigma;\Gamma;\Delta, \mathtt{this} : \mathsf{var}(I, s_1), \mathtt{value} : \mathsf{var}(\mathsf{int}, s_2) \vdash e : B_s$. *If* $\Sigma \vdash (I', s_1') <: (I, s_1)$ *and* $s_2' \sqsubseteq s_2$ *and*

$$\forall q \in (\mathrm{dom}\,(\Gamma(I)) \cap (\mathsf{FNames} \cup \{\mathtt{balance}\})).\Gamma(I)(q) = \Gamma(I')(q),$$

then $\Sigma;\Gamma;\Delta, \mathtt{this} : \mathsf{var}(I', s_1'), \mathtt{value} : \mathsf{var}(\mathsf{int}, s_2') \vdash e : B_s$.

Lemma 6. *Let* $\Sigma;\Gamma;\Delta, \mathtt{this} : \mathsf{var}(I, s_1), \mathtt{value} : \mathsf{var}(\mathsf{int}, s_2) \vdash S : \mathsf{cmd}(s)$. *If* $\Sigma \vdash (I', s_1') <: (I, s_1)$ *and* $s_2' \sqsubseteq s_2$ *and*

$$\forall q \in (\mathrm{dom}\,(\Gamma(I)) \cap (\mathsf{FNames} \cup \{\mathtt{balance}\})).\Gamma(I)(q) = \Gamma(I')(q),$$

then $\Sigma;\Gamma;\Delta, \mathtt{this} : \mathsf{var}(I', s_1'), \mathtt{value} : \mathsf{var}(\mathsf{int}, s_2') \vdash S : \mathsf{cmd}(s)$.

Our next theorem is then a preservation theorem for statements S and memory states env_{SV}. However, because of the dual nature of our types, the theorem also must express two different kinds of preservation; namely (1) preservation of well-typedness of the resulting memory state env'_{SV} after the execution of S, and (2) preservation of the *values* stored in the memory in all entries that have a security level that is not greater than or equal to the level of the command S. The proof is by induction on the derivation of $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to_{\mathrm{S}} \mathsf{env}'_{SV}$ and can be found in Appendix E.

Theorem 3 (Preservation). *Assume that* $\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s)$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_T$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_{SV}$, *and* $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to_{\mathrm{S}} \mathsf{env}'_{SV}$. *Then,* $\Sigma;\Gamma;\Delta \vdash \mathsf{env}'_{SV}$ *and* $\Gamma;\Delta \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}'_{SV}$ *for every* s' *such that* $s \not\sqsubseteq s'$.

The Preservation theorem assures us that the promise made by the type $\mathsf{cmd}(s)$ is actually fulfilled: if S is typed as $\mathsf{cmd}(s)$, then every variable or field written to in S will be of level s *or higher*; hence every variable or field of a level that is *strictly lower* than or incomparable to s will be unaffected. In other words, in addition to well-typedness, what is shown to be 'preserved' in this theorem is the *values* at levels lower than or incomparable to s.

Finally, we can use the preceding theorems to show soundness for the type system. The soundness theorem expresses that, if a statement S is well-typed *to any level* s_1 and we execute S with any two states env^1_{SV} and env^2_{SV} that agree *up to any level* s_2, then the resulting states $\mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^{2'}_{SV}$ will still agree on all values up to level s_2. This ensures noninterference, since any difference in values of a higher level than s_2 cannot induce a difference in the computation of values at any lower levels.

Theorem 4 (Soundness). *Assume that* $\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s_1)$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}_T$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}^1_{SV}$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}^2_{SV}$, $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, $\mathsf{env}_T \vdash \langle S, \mathsf{env}^1_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV}$, *and* $\mathsf{env}_T \vdash \langle S, \mathsf{env}^2_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV}$. *Then,* $\Gamma;\Delta \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$.

Proof Sketch. There are two cases to consider. If $s_1 \not\sqsubseteq s_2$ (i.e., s_2 is either strictly below s_1, or they are incomparable), then, by Theorem 3, we can conclude that $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^{1'}_{SV}$ and $\Gamma;\Delta \vdash \mathsf{env}^2_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$; as we know that $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, it therefore also holds that $\Gamma;\Delta \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$. If $s_1 \sqsubseteq s_2$, since S may alter variables up to level s_1, all variables up to level s_2 that are also up to level s_1 have the same value in env^1_{SV} and env^2_{SV}, so they also have the same value within $\mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^{2'}_{SV}$. The proof is by induction on the sum of the lengths of the inferences for $\mathsf{env}_T \vdash \langle S, \mathsf{env}^1_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV}$, and $\mathsf{env}_T \vdash \langle S, \mathsf{env}^2_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV}$. The key observation is that the choice of the last rule in both inferences is syntax-driven (according to the outmost operator in S); the only case when the syntax does not univocally identifies the rule to use is when S is a `while`. So, we preliminarily prove that also in this case no uncertainty is possible. Indeed, let $\mathsf{env}^1_{SV} \vdash e \to_{\mathsf{e}} b_1$ and $\mathsf{env}^2_{SV} \vdash e \to_{\mathsf{e}} b_2$, where $b_1, b_2 \in \{\mathsf{T}, \mathsf{F}\}$. Since we are assuming that $s_1 \sqsubseteq s_2$ and by hypothesis $\Gamma \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, we can conclude that $b_1 = b_2$ by Theorem 2. Hence, both inferences terminate either with [WHILE$_{\mathsf{F}}$] or [WHILE$_{\mathsf{T}}$], according to the evaluation of the guard. We thus have proved that both inferences terminate with the very same rule of Figure 6; the proof then consists in considering the last rule in these inferences (the details are in Appendix F). □

Theorem 4 corresponds to the soundness theorem proved by Volpano et al. [53] for their While-like language. However, given the class-based nature of TinySol, we can actually take this result one step further and allow even parts of the code to vary. Specifically, given two 'method table' environments, env^1_T and env^2_T, we just require that these two environments agree up to the same level s_2 to ensure agreement of the resulting two states $\mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^{2'}_{SV}$. We state this in the following theorem.

Theorem 5 (Extended soundness). *Assume that* $\Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s_1)$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}^1_T$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}^2_T$, $\Gamma \vdash \mathsf{env}^1_T =_{s_2} \mathsf{env}^2_T$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}^1_{SV}$, $\Sigma;\Gamma;\Delta \vdash \mathsf{env}^2_{SV}$, $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, $\mathsf{env}^1_T \vdash \langle S, \mathsf{env}^1_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV}$, *and* $\mathsf{env}^2_T \vdash \langle S, \mathsf{env}^2_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV}$. *Then,* $\Gamma;\Delta \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$.

Proof. By induction on the sum of the lengths of the inferences for $\mathsf{env}^1_T \vdash \langle S, \mathsf{env}^1_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV}$, and $\mathsf{env}^2_T \vdash \langle S, \mathsf{env}^2_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV}$. All the cases are like the corresponding ones in the proof of Theorem 4, except those for method invocations (that are the only ones where env_T plays a role). We only consider the case in which [CALL] was the last one used in the inferences; the case for [DCALL] is similar. There are two cases to consider:

—If $s_1 \not\sqsubseteq s_2$, then env^1_T and env^2_T only agree up to level s_2, but may differ on other levels, including s_1 (which is either strictly above s_2, or they are incomparable). Thus the call may be to two different methods (albeit with the same signature), namely $\mathsf{env}^1_T(Y)(f) = (\tilde{x}, S_1)$ and $\mathsf{env}^2_T(Y)(f) = (\tilde{x}, S_2)$. However, as S is typed to level s_1, this cannot induce a difference at any level lower than or incomparable to s_1. In particular, by the premises of rule [CALL] we have that $\mathsf{env}^1_T \vdash \langle S_1, \mathsf{env}^1_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV}$ and $\mathsf{env}^2_T \vdash \langle S_2, \mathsf{env}^2_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV}$. By two applications of Theorem 3, we can then conclude that $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^{1'}_{SV}$ and $\Gamma;\Delta \vdash \mathsf{env}^2_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$. As we know by assumption that $\Gamma;\Delta \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, we can therefore also conclude that $\Gamma;\Delta \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$.

—If $s_1 \sqsubseteq s_2$, then env_T^1 and env_T^2 agree on all levels up to level s_2 *including* level s_1. By assumption, $\Gamma \vdash \mathsf{env}_T^1 =_{s_2} \mathsf{env}_T^2$ and so, by [EQ-ENV$_T$], $\mathsf{env}_T^1(Y) = \mathsf{env}_T^2(Y)$, since Y resides at level s_1; in particular, $\mathsf{env}_T^1(Y)(f) = \mathsf{env}_T^2(Y)(f) = (\tilde{x}, S')$. The rest of the proof then proceeds like in the case for [CALL] of Appendix F. □

4.5 Extending the Type System to Transactions

A transaction is nothing but a sequence of method calls with parameters that are values and with `sender` that is set to an account address, which corresponds to a minimal implementation of $I^\top$. Thus, the theorems from the preceding section can easily be extended to transactions and blockchains.

A blockchain consists of a set of contract declarations DC, followed by a transaction τ. Hence, we can conclude $\Sigma; \Gamma; \Delta \vdash DC\ \tau : \mathsf{cmd}(s)$, if it holds that $\Sigma; \Gamma; \Delta \vdash DC$ and $\Sigma; \Gamma; \Delta \vdash \tau : \mathsf{cmd}(s)$. The latter can be simply concluded by the following rules:

$$[\text{T-EMPTY}]\ \frac{}{\Sigma; \Gamma; \Delta \vdash \epsilon : \mathsf{cmd}(s)} \qquad [\text{T-TRANS}]\ \frac{\Sigma; \Gamma; \Delta' \vdash Y\,.\,f(\tilde{v})\$z : \mathsf{cmd}(s) \quad \Sigma; \Gamma; \Delta \vdash \tau : \mathsf{cmd}(s)}{\Sigma; \Gamma; \Delta \vdash A \overset{z}{\rightsquigarrow} Y\,.\,f(\tilde{v}), \tau : \mathsf{cmd}(s)}$$

where:

$$\Delta' = \mathtt{this} : \mathsf{var}(\Gamma(A))$$

This gives us the following two results:

Lemma 7. *If* $\Sigma; \Gamma; \Delta \vdash DC$ *and* $\left\langle DC, \mathsf{env}_{ST}^{\emptyset} \right\rangle \to_{\text{DC}} \mathsf{env}_{ST}$, *then* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{ST}$.

Lemma 8. *If* $\Sigma; \Gamma; \Delta \vdash A \overset{z}{\rightsquigarrow} Y\,.\,f(\tilde{v}), \tau : \mathsf{cmd}(s)$ *and* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{ST}$ *and* $\left\langle A \overset{z}{\rightsquigarrow} Y\,.\,f(\tilde{v}); \tau, \mathsf{env}_{ST} \right\rangle \to_{\text{B}} \langle \tau, \mathsf{env}'_S; \mathsf{env}_T \rangle$, *then also* $\Sigma; \Gamma; \Delta \vdash \mathsf{env}'_S$.

As the initial step (the 'genesis event') does nothing except transforming the declaration DC into the environment representation env_{ST}, the first result is obvious. As the rule [TRANS] just unwraps a transaction step into a call to the corresponding method, the second result follows directly from the Preservation theorem; this can then be generalised in an obvious way to the whole transaction list. Likewise, the Safety and Soundness theorems can be extended to transactions in the same manner.

4.6 Noninterference and Call Integrity

As immediately evident from Definition 6 and Theorem 4, well-typedness ensures noninterference:

Corollary 1 (Noninterference). *Assume a set of security levels* $\mathcal{S} \triangleq \{ L, H \}$, *with* $L \sqsubseteq L \sqsubseteq H \sqsubseteq H$, *and furthermore that* $\Sigma; \Gamma; \Delta \vdash \tau : \mathsf{cmd}(s)$, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{ST}^1$, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{ST}^2$, $\Gamma \vdash \mathsf{env}_{ST}^1 =_L \mathsf{env}_{ST}^2$, $\langle \tau, \mathsf{env}_{ST}^1 \rangle \to_{\text{B}}^+ \mathsf{env}_{ST}^{1'}$, *and* $\langle \tau, \mathsf{env}_{ST}^2 \rangle \to_{\text{B}}^+ \mathsf{env}_{ST}^{2'}$. *Then,* $\Gamma \vdash \mathsf{env}_{ST}^{1'} =_L \mathsf{env}_{ST}^{2'}$.

Proof. By Theorem 4 and Lemma 8. □

From Corollary 1, we then obviously also have that $\Gamma \vdash \mathsf{env}_S^{1'} =_L \mathsf{env}_S^{2'}$, regardless of whether s is L or H. In particular, we can assign security levels to entire contracts, as well as all their members. Thus, our type system can be used to ensure noninterference according to Definition 6.

As we previously argued in Remark 2, noninterference and call integrity are incomparable properties. However, as our next theorem shows, well-typedness actually also ensures call integrity. This is surprising, so before stating the theorem, we should give some hints as to why this is the case.

The definition of call integrity (Definition 5) requires the execution of any code in a contract X to be unaffected by all contracts in an 'untrusted set' $\mathcal{Y}$, regardless of whether parts of the code in

$\mathcal{Y}$ execute before, meanwhile or after the code in X. This is expressed by a quantification over all possible traces resulting from a change in $\mathcal{Y}$, i.e., either in the code or in the values of the fields. Regardless of any such change, it must hold that the sequence of method calls originating from X be the same.

Noninterference, on the other hand, says nothing about execution traces, but only speaks of the correspondence between values residing in the memory before and after the execution step. The two counter-examples used in Remark 2 made use of this fact:

- The first counter-example had X be unable to perform any method call, thus obviously satisfying call integrity, but allowed different `balance` values to be transferred into it from a 'high' context by means of a method call, thereby violating noninterference. However, this situation is ruled out by well-typedness (cf. rule [T-CALL]), because well-typedness disallows *any* method calls from a 'high' to a 'low' context, precisely because every method call may transfer the `value` parameter along with each call.
- The second counter-example had an `if` statement in X (the 'low' context) depend on a field value in a 'high' context. The two branches then perform two different method calls, thus enabling a change of the 'high' context to induce two different execution traces for X. Thus, the example satisfies noninterference, because no value stored in the memory is changed, but it obviously does not satisfy call integrity. However, this situation is also ruled out by well-typedness, because [T-IF] does not allow the boolean guard expression e in a 'low' context to depend on a value from a 'high' context. In particular, the guard of the `if` is a test on the `balance` of contract Y, which is high (i.e., untrusted); so, the only possibility for typing the body of go is to try with $\mathsf{cmd}(H)$ that, however, cannot be assigned to a method of X (which is low, i.e., trusted).

Thus, both of the two counter-examples would be rejected by the type system. With a setting of L for the 'trusted' segment and H for the 'untrusted', no values or computations performed in the untrusted segment can affect the values in the trusted segment, nor the value of any expression in this segment, nor can it even perform a call into the trusted segment. On the other hand, the trusted segment *can* call out into the untrusted part, but such a call cannot then reenter the trusted segment: it must return before any further calls from the trusted segment can happen.

THEOREM 6 (WELL-TYPEDNESS IMPLIES CALL INTEGRITY). *Let $\mathcal{S} \triangleq \{ L, H \}$ with $L \sqsubseteq H$. Choose a bipartition of* ANames *into $\mathcal{X}$ and $\mathcal{Y}$ and fix a type assignment Γ such that*

- $\forall X \in \mathcal{X} \,.\, \Gamma(X) = I_L$, *for some I such that*
  - $\forall q \in (\mathrm{dom}\,(\Gamma(I)) \cap (\mathsf{FNames} \cup \{\mathtt{balance}\}))\ \exists B \,.\, \Gamma(I)(q) = \mathsf{var}(B_L)$, *and*
  - $\forall f \in (\mathrm{dom}\,(\Gamma(I)) \cap \mathsf{MNames})\ \exists \widetilde{B_{s'}} \,.\, \Gamma(I)(f) = \mathsf{proc}(\widetilde{B_{s'}}) : L$
- Γ *assigns level H to all other interfaces, fields and methods.*

Also assume that $\Sigma; \Gamma; \Delta \vdash \tau : \mathsf{cmd}(s)$, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}^1_{ST}$, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}^2_{ST}$, $\Gamma \vdash \mathsf{env}^1_{ST} =_L \mathsf{env}^2_{ST}$, $\langle \tau, \mathsf{env}^1_{ST} \rangle \xrightarrow{\pi_1}{}^*_{\mathrm{B}}\ \mathsf{env}^{1'}_{ST}$, *and* $\langle \tau, \mathsf{env}^2_{ST} \rangle \xrightarrow{\pi_2}{}^*_{\mathrm{B}}\ \mathsf{env}^{2'}_{ST}$. *Then, $\pi_1 \downarrow_X = \pi_2 \downarrow_X$, for any $X \in \mathcal{X}$.*

PROOF. By induction on the length of the transaction τ. The base case (viz., $\tau = \epsilon$) is trivial, since $\pi_1 = \pi_2 = \epsilon$. For the inductive case, let $\tau = A \overset{z}{\rightsquigarrow} Y \,.\, f(\tilde{v}), \tau'$. We consider two possible cases.

If $Y \in \mathcal{Y}$, then the result is immediate by [T-CALL], because no call into $\mathcal{X}$ is allowed due to the implicit write to `balance` in every method call. Thus, neither of the traces π_1 or π_2 would contain any method call from any X in $\mathcal{X}$, and therefore $\pi_1 \downarrow_X = \pi_2 \downarrow_X$ obviously holds since $\pi_1 \downarrow_X = \epsilon = \pi_2 \downarrow_X$.

If $Y \in \mathcal{X}$, we know by assumption that $\Gamma(Y) = I_L$ for some I; hence $\mathsf{env}_T^1(Y) = \mathsf{env}_T^2(Y) = \mathsf{env}_M^Y$ and the method called is the same in both runs, say $\mathsf{env}_M^Y(f) = (\tilde{x}, S)$. Let $\mathsf{env}_V = \{(\tilde{x}, \tilde{v}), (\mathtt{this}, Y), (\mathtt{sender}, A), (\mathtt{value}, z)\}$; then,

$$\pi_1 = A \overset{z}{\rightsquigarrow} Y.f(\tilde{v}), \pi_1', \pi_1'' \qquad \pi_2 = A \overset{z}{\rightsquigarrow} Y.f(\tilde{v}), \pi_2', \pi_2''$$

where

$$\mathsf{env}_T^1 \vdash \langle S, \mathsf{env}_S^1, \mathsf{env}_V \rangle \xrightarrow{\pi_1'}_{\mathrm{S}} \mathsf{env}_S^{1''}, \mathsf{env}_V^{1'} \qquad \mathsf{env}_T^2 \vdash \langle S, \mathsf{env}_S^2, \mathsf{env}_V \rangle \xrightarrow{\pi_2'}_{\mathrm{S}} \mathsf{env}_S^{2''}, \mathsf{env}_V^{2'}$$

and

$$\left\langle \tau', \mathsf{env}_S^{1''}, \mathsf{env}_T^1 \right\rangle \xrightarrow{\pi_1''}_{\mathrm{B}}{}^{*} \mathsf{env}_{ST}^{1'} \qquad \left\langle \tau', \mathsf{env}_S^{2''}, \mathsf{env}_T^2 \right\rangle \xrightarrow{\pi_2''}_{\mathrm{B}}{}^{*} \mathsf{env}_{ST}^{2'}$$

Since $\Gamma \vdash \mathsf{env}_S^{1''} =_L \mathsf{env}_{ST}^{2''}$ holds by Theorem 5, we can use the induction hypothesis to obtain that $\pi_1'' \downarrow_X = \pi_2'' \downarrow_X$, for every $X \in \mathcal{X}$. To prove that $\pi_1' \downarrow_X = \pi_2' \downarrow_X$, we prove the following claim:

$$\begin{aligned} &\text{If } \Sigma;\Gamma;\Delta \vdash S : \mathsf{cmd}(s) \text{ and } \Sigma;\Gamma;\Delta \vdash \mathsf{env}_{ST}^1 \text{ and} \\ &\quad \Sigma;\Gamma;\Delta \vdash \mathsf{env}_{ST}^2 \text{ and } \Gamma \vdash \mathsf{env}_{ST}^1 =_L \mathsf{env}_{ST}^2 \text{ and} \\ &\quad \mathsf{env}_T^1 \vdash \langle S, \mathsf{env}_S^1, \mathsf{env}_V \rangle \xrightarrow{\pi_1'}_{\mathrm{S}} \mathsf{env}_S^{1''}, \mathsf{env}_V^{1'} \text{ and} \\ &\quad \mathsf{env}_T^2 \vdash \langle S, \mathsf{env}_S^2, \mathsf{env}_V \rangle \xrightarrow{\pi_2'}_{\mathrm{S}} \mathsf{env}_S^{2''}, \mathsf{env}_V^{2'} \\ &\text{then } \pi_1' \downarrow_X = \pi_2' \downarrow_X, \text{ for any } X \in \mathcal{X}. \end{aligned} \tag{3}$$

This claim is proved by induction on the inference of the transitions in (3). First of all, by Theorem 5 we know that $\Gamma \vdash \mathsf{env}_S^{1'}, \mathsf{env}_V^{1'} =_L \mathsf{env}_S^{2'}, \mathsf{env}_V^{2'}$. Hence, like in the proof of Theorem 4, the last rule used in both inferences is the same: in all cases, this is syntax-driven and, whenever there is a guard that may change the evolution, this guard is evaluated at the same value, because of Theorem 2. There are four base cases: [SKIP], [ASSV], [ASSF] and [WHILE$_\mathrm{F}$]. All cases are trivial, since no call is performed and so $\pi_1' = \epsilon = \pi_2'$. There are six inductive cases, according to the last rule used in the inferences:

- [DECV]: Then $S = \mathsf{var}(B_{s'})\ x\ \mathtt{:=}\ e\ \mathtt{in}\ S'$. By Theorem 2, $\mathsf{env}_S^1, \mathsf{env}_V \vdash e \to_{\mathrm{e}} v$ and $\mathsf{env}_S^2, \mathsf{env}_V \vdash e \to_{\mathrm{e}} v$. Then,

$$\mathsf{env}_T^1 \vdash \langle S', \mathsf{env}_S^1, \{(x, v)\} \cup \mathsf{env}_V \rangle \xrightarrow{\pi_1'}_{\mathrm{S}} \mathsf{env}_S^{1'}, \{(x, v_1)\} \cup \mathsf{env}_V^{1'}$$
$$\mathsf{env}_T^2 \vdash \langle S', \mathsf{env}_S^2, \{(x, v)\} \cup \mathsf{env}_V \rangle \xrightarrow{\pi_2'}_{\mathrm{S}} \mathsf{env}_S^{1'}, \{(x, v_2)\} \cup \mathsf{env}_V^{2'}$$

  and, by the induction hypothesis on (3), we conclude that $\pi_1' \downarrow_X = \pi_2' \downarrow_X$, for all $X \in \mathcal{X}$.
- [SEQ]: Then $S = S_1; S_2$ and so $\pi_1' = \pi_1^1, \pi_1^2$ and $\pi_2' = \pi_2^1, \pi_2^2$. By two applications of the induction hypothesis on (3), we obtain that $\pi_1^1 \downarrow_X = \pi_2^1 \downarrow_X$ and $\pi_1^2 \downarrow_X = \pi_2^2 \downarrow_X$, that allow us to easily conclude.
- [IF]: Then, $S = \mathtt{if}\ e\ \mathtt{then}\ S_\mathsf{T}\ \mathtt{else}\ S_\mathsf{F}$. By Theorem 2, $\mathsf{env}_S^1, \mathsf{env}_V \vdash e \to_{\mathrm{e}} b$ and $\mathsf{env}_S^2, \mathsf{env}_V \vdash e \to_{\mathrm{e}} b$; so, the same branch is chosen in both contexts and the same command S_b is executed. The case then holds for S_b by the induction hypothesis on (3).
- [WHILE$_\mathrm{T}$]: The argument is the same as for [IF] above.
- [CALL]: Then $S = e_1.g(\tilde{e})\$e_2$. By Theorem 2, $\mathsf{env}_S^1, \mathsf{env}_V \vdash e_1 \to_{\mathrm{e}} Z$ and $\mathsf{env}_S^2, \mathsf{env}_V \vdash e_1 \to_{\mathrm{e}} Z$, for some address Z such that $\Gamma(Z) = I_s$ for some interface I and security level s. Moreover, $\mathsf{env}_T^1(Z)(g) = (\widetilde{x_1}, S_1)$ and $\mathsf{env}_T^2(Z)(g) = (\widetilde{x_2}, S_2)$, for $|\widetilde{x_1}| = |\widetilde{x_2}| = |\tilde{e}|$. Moreover, $\tilde{e}$ and e_2 are both typed as L and, by Theorem 2, are evaluated at the same values $\widetilde{v'}$ and z'. There are two cases, depending on the level s:

—If $s = L$, then $S_1 = S_2 = S'$ and $\widetilde{x_1} = \widetilde{x_2} = \widetilde{x'}$. Now, let $\mathsf{env}'_V = \{(\widetilde{x'}, \widetilde{v'}), (\mathtt{this}, Z), (\mathtt{sender}, Y), (\mathtt{value}, z')\}$; then,

$$\mathsf{env}^1_T \vdash \langle S', \mathsf{env}^1_S, \mathsf{env}'_V\rangle \xrightarrow{\hat{\pi}_1}_{\mathrm{S}} \mathsf{env}^{1'}_S, \mathsf{env}^{1'}_V \qquad \mathsf{env}^2_T \vdash \langle S', \mathsf{env}^2_S, \mathsf{env}'_V\rangle \xrightarrow{\hat{\pi}_2}_{\mathrm{S}} \mathsf{env}^{1'}_S, \mathsf{env}^{2'}_V$$

By the induction hypothesis on (3), $\hat{\pi}_1 \downarrow_X = \hat{\pi}_2 \downarrow_X$, for any $X \in \mathcal{X}$; so,

$$\pi'_1 \downarrow_X = \left(Y \overset{z'}{\rightsquigarrow} Z.g(\widetilde{v'}), \hat{\pi}_1\right) \downarrow_X = \left(Y \overset{z'}{\rightsquigarrow} Z.g(\widetilde{v'}), \hat{\pi}_2\right) \downarrow_X = \pi'_2 \downarrow_X$$

—If $s = H$, then S_1 and S_2 may differ. However, if $s = H$, then by definition $Z \in \mathcal{Y}$ and so we can reason like in the case where $Y \in \mathcal{Y}$, at the beginning of this proof. In particular, by letting $\hat{\pi}_1$ and $\hat{\pi}_2$ be the traces generated by S_1 and S_2 respectively, we know that $\hat{\pi}_1 \downarrow_X = \epsilon = \hat{\pi}_2 \downarrow_X$, for all $X \in \mathcal{X}$, because the call can never *reenter* any contract in the 'low' segment $\mathcal{X}$; in particular, no further call from X can appear in the traces. Hence,

$$\pi'_1 \downarrow_X = \left(Y \overset{z'}{\rightsquigarrow} Z.g(\widetilde{v'})\right) \downarrow_X = \pi'_2 \downarrow_X$$

[DCALL]: Then $S = \mathtt{dcall}\ e.g(\tilde{e})$. Like in the previous case, e and $\tilde{e}$ are both typed as L and, by Theorem 2, are evaluated at the same values Z and $\widetilde{v'}$; moreover, $\Gamma(Z) = I_s$, $\mathsf{env}^1_T(Z)(g) = (\widetilde{x_1}, S_1)$ and $\mathsf{env}^2_T(Z)(g) = (\widetilde{x_2}, S_2)$, for $|\widetilde{x_1}| = |\widetilde{x_2}| = |\tilde{e}|$. The proof then proceeds like in the previous case, with $z' = 0$. □

Theorem 6 tells us that *every* contract in the trusted segment $\mathcal{X}$ has call integrity w.r.t. the untrusted segment $\mathcal{Y}$. This is thus a stronger condition than that of Definition 5, which only defines call integrity for a *single* contract $X \in \mathcal{X}$. This means that our type system will reject cases where, e.g., X calls another contract $Z \in \mathcal{X}$ and Z calls `send()` methods of different contracts, depending on a 'high' value. As `send()` is always ensured to do nothing, such calls could never lead to X being reentered, so this would actually still be safe, even though Z itself would not satisfy call integrity. Thus, this is an example of what resides in the 'slack' of our type system. However, this situation seems rather contrived, since it depends specifically on the `send()` method, which is always ensured to do nothing except returning. For practical purposes, it would be strange to imagine a contract $X \in \mathcal{X}$ having call integrity w.r.t. $\mathcal{Y}$, but *without* the other contracts in $\mathcal{X}$ also satisfying call integrity w.r.t. $\mathcal{Y}$. Thus, our type system seems to yield a reasonable approximation to the property of call integrity.

Remark 4. As we have previously remarked, the side condition $s \sqsubseteq s_2, s_3$ for method calls is instrumental in ensuring that well-typedness also ensures call-integrity, yet in the case of delegate calls this side condition is not needed. The absence of this side condition means that an untrusted (i.e., High) contract, say Y, *can* perform a *delegate* call into the trusted (i.e., Low) segment in a well-typed program, say, to a contract X, unlike the case for ordinary method calls. However, this does not violate the call-integrity property because, in the case of delegate calls, the code is executed in the context of the *caller* contract. The call trace would register a call from Y to X, e.g.,

$$\mathsf{env}_T \vdash \langle \mathtt{dcall}\ Y.f(\tilde{e}), \mathsf{env}_{SV}\rangle \xrightarrow{Y \overset{0}{\rightsquigarrow} X.f(\tilde{v}), \pi}_{\mathrm{S}} \mathsf{env}'_S, \mathsf{env}_V$$

but the magic variables `this` and `sender` are *not* rebound in a delegate call, so if the body of f performs any further calls, say to a method g in a contract Z, then that call would *also* be registered in the call trace with Y as the sender[7]. The control flow does not actually reenter the trusted segment, so well-typedness still implies call-integrity. ∎

[7]Of course, for such a call to be well-typed, it would also be necessary that both the delegate call to $X.f$ and the method call to $Z.g$ were typable as $\mathsf{cmd}(H)$, to permit the delegate call from High to Low.

To conclude the presentation of the type system, we remark that it only approximates noninterference and call integrity. Indeed, these are properties defined on top of the operational semantics of the language, whereas the type system is a static analysis technique and so it cannot consider the dynamic evolution of the memory. Consider

$$\texttt{x := T ; if x then } S_{\text{good}} \texttt{ else } S_{\text{bad}}$$

where S_{good} and S_{bad} are two commands that behave, respectively, well and badly at runtime w.r.t. both noninterference and call integrity. Clearly, the program behaves well w.r.t. our two reference properties, because it always takes the then-branch; nevertheless, it is rejected by the type checker when it discovers that there is a part of code that does not respect the typing constraints (even though this part will never be executed at runtime).

5 Examples and Limitations

Let us see a few examples of the application of the type system. To begin, consider the first counter-example in Remark 2, which should be ill-typed by the type system. In the counter-example we say that X is Low and Y is High, so we let them respectively implement the interfaces IL and IH defined as follows:

```
interface IL : I^⊤ {          interface IH : I^⊤ {          contract X : IL_L { ...
  balance : var(int, L)         balance : var(int, H)         }
  send    : proc():L            send    : proc():H
  go      : proc():L            go      : proc():H          contract Y : IH_H { ...
}                             }                             }
```

Hence, we can use $\Sigma = \{(\texttt{IL}, \texttt{I}^\top), (\texttt{IH}, \texttt{I}^\top), (\texttt{I}^\top, \texttt{I}^\top)\}$ and Γ be

$$\begin{array}{lll} \texttt{X} & : & \texttt{IL}_L \\ \texttt{Y} & : & \texttt{IH}_H \end{array} \qquad\qquad \begin{array}{lll} \texttt{IL} & : & \{(\texttt{balance}, \text{var}(\text{int}, L)), (\texttt{send}, \text{proc}():L), (\texttt{go}, \text{proc}():L)\} \\ \texttt{IH} & : & \{(\texttt{balance}, \text{var}(\text{int}, H)), (\texttt{send}, \text{proc}():H), (\texttt{go}, \text{proc}():H)\} \end{array}$$

A part of the failing typing derivation for the body of the method Y.go() in the declaration of contract Y (thus, when using rules [T-DEC-C] and [T-DEC-M], invoked with an empty variable environment) is:

$$\dfrac{\dfrac{\dfrac{\Delta(\texttt{this}) = \text{var}(\texttt{IH}_H)}{\Sigma;\Gamma;\Delta \vdash \texttt{this} : \texttt{IH}_H} \quad \Gamma(\texttt{IH})(\texttt{balance}) = \text{var}(\text{int}_H)}{\Sigma;\Gamma;\Delta \vdash \texttt{this.balance} : \text{int}_H} \quad \dfrac{\textsc{TypeOf}_\Gamma(\texttt{X}) = \Gamma(\texttt{X}) = \texttt{IL}_L}{\Sigma;\Gamma;\Delta \vdash \texttt{X} : \texttt{IL}_L} \quad \Gamma(\texttt{IL})(\texttt{go}) \neq \text{proc}():H}{\Sigma;\Gamma;\Delta \nvdash \texttt{X.go()\$this.balance} : \text{cmd}(H)} \tag{4}$$

where $\Delta = \{(\texttt{this}, \text{var}(\texttt{IH}_H)), (\texttt{value}, \text{var}(\text{int}, H)), (\texttt{sender}, \text{var}(I^\top, s_\top))\}$ and the inference has been obtained by using rules [T-CALL], [T-FIELD], and two applications of [T-VAR]. We have that $\Sigma;\Gamma;\Delta \vdash \texttt{this.balance} : \text{int}_H$ in contract Y, so in order for the declaration of method go() in Y to be well-typed, the body of the method must be typable as $\text{proc}():H$ by rule [T-DEC-M]. However, as the derivation in (4) illustrates, this constraint cannot be satisfied, because the lookup $\Gamma(\texttt{IL})(\texttt{go})$ yields $\text{proc}():L$, that cannot be promoted to $\text{proc}():H$ through subtyping, because the $\widetilde{\text{proc}(B_{s'}):s}$ type constructor is contravariant in s.

The above example is simple, since the name X is 'hard-coded' directly in the body of Y.go(), and therefore the type check fails already while checking the contract definition. However, suppose X were instead received as a parameter. Then the signature of the method Y.go would have instead to be

$$\texttt{go} \ : \ \text{proc}(\texttt{IH}_H):H$$

```
contract Fund {
  mapping(address => uint) shares;
  function withdraw() {
    if (msg.sender.send(shares[msg.sender]))
      shares[msg.sender] = 0;
  }
}

contract Bob {
  bool sent = false;
  function ping(address c) {
    if (!sent) {
      sent = true;
      c.call.value(2) ();
    }
  }
}
```

Fig. 18. The false positive and false negative examples from [26].

and the type check would then fail at the call-site, if a Low address were passed. The following shows a part of the failing typing derivation for the call `Y.go(X)$this.balance`, where the parameter `X` is assumed to implement the interface `IL` as before (the typing of `this.balance` is like in the leftmost part of (4)):

$$\dfrac{\cdots\quad \dfrac{\textsc{TypeOf}_{\Gamma}(\mathtt{Y}) = \Gamma(\mathtt{Y}) = \mathtt{IH}_H}{\Sigma;\Gamma;\Delta \vdash \mathtt{Y} : \mathtt{IH}_H} \qquad \Gamma(\mathtt{IH})(\mathtt{go}) = \mathsf{proc}(\mathtt{IH}_H) : H \qquad \dfrac{\dfrac{\textsc{TypeOf}_{\Gamma}(\mathtt{X}) = \Gamma(\mathtt{X}) = \mathtt{IL}_L}{\Sigma;\Gamma;\Delta \vdash \mathtt{X} : \mathtt{IL}_L} \qquad \dfrac{\Sigma(\mathtt{IL}) = \mathtt{I}^{\top} \not= \mathtt{IH}}{\Sigma \not\vdash \mathtt{IL} <: \mathtt{IH}}}{\Sigma;\Gamma;\Delta \not\vdash \mathtt{X} : \mathtt{IH}_H}}{\Sigma;\Gamma;\Delta \not\vdash \mathtt{Y.go(X)\$this.balance} : \mathsf{cmd}(H)} \tag{5}$$

This has been obtained by rule [T-CALL] and by two applications of [T-VAR]. The method call expects a parameter of type $\mathtt{IH}_H$, but $\mathtt{IL}_L$ cannot be coerced up to $\mathtt{IH}_H$ through subtyping, because of the interfaces definition (`IH` and `IL` are unrelated in the interface inheritance tree, since they are both children of $\mathtt{I}^{\top}$).

Thus, the type system prevents calls from High to Low, regardless of whether the Low address is 'hard-coded' or passed as a parameter to a High method. However, the aforementioned examples also illustrate a limitation of our type system approach to ensuring call integrity: the entire blockchain must be checked, i.e., both all contracts *and* the whole transaction. This is necessary since the type check can fail at the call-site of a method, as in the example shown in (5), and the call-site of any method can be a transaction.

Next, we shall briefly consider two examples, a false positive and a false negative one, reported by Grishchenko et al. in [26, Figures 2.e and 2.f]; for completeness, we report their code here in Figure 18 (even if written in Solidity and not in TinySol, the reader should understand their meaning). These are Solidity contracts that are misclassified w.r.t. reentrancy by the static analyser Oyente [33]; this is a symbolic execution tool exclusively designed to analyze Ethereum smart contracts, by following their execution model and directly working with the EVM bytecode, without access to the high level representation, e.g., Solidity.

```
contract X : IBank_L {
 field owner = A;
 transfer(recipient, amount) {
  if this.sender = this.owner then
     recipient.deposit(this.sender)$amount
  else skip
 }
  ...
}
```

```
contract Y : IBank_H {
 field credit = 0;
 deposit(owner) {
   this.credit = this.value;
   this.owner = owner
 }
   ...
}
```

Fig. 19. A two-bank setup.

The false negative example is the first contract in Figure 18 and it relies on a misplaced update of a field value[8]: indeed, the invocation of `msg.sender.send` in line 4 uses `shares[msg.sender]` *before* its update, that happens in line 5; so, in principle, the recipient can call back the function again and again, always with the same value of `shares[msg.sender]`. This is similar to what happens in the example of Figure 1. In this example, suppose `X` were assigned the level L and `Y` the level H. With a transaction $A \overset{z}{\rightsquigarrow}$ `X.transfer(Y)` (for any address A and any amount of currency z), the type system would then correctly reject this blockchain because of the inherent flow from High to Low that is implicit in the call `X.transfer(this)` issued by `Y`. The typing derivation would fail in a way similar to the situation depicted in (5).

The false positive example of Grishchenko et al. from [26] is the second contract in Figure 18 and it is also similar to the example in Figure 1, but this time just with the assignment to the guard variable correctly placed *before* the method call (i.e., with lines 6 and 7 switched in Figure 1). This too would be rejected by our type system, since it does not take the ordering of statements in sequential composition into account (i.e., rule [T-SEQ]). Thus, this example constitutes a false positive for our type system as well, which is hardly surprising.

Let us now consider a true positive example. Figure 19 illustrates a part of the code for two banks, which would allow users to store some of their assets and also to transfer assets between them[9]. We assume both banks implement the same interface `IBank` (so, they must have the same fields and methods—even though we only explicitly write the relevant ones and hide the remainder in the dots '...'), but with different security settings: `X` is L and `Y` is H, meaning the latter is untrusted. There is no callback from `Y`, so in this setup a blockchain with a transaction `A` $\overset{0}{\rightsquigarrow}$ `X.transfer(Y,1)` would actually be accepted by the type system, because the Low values from `X` can safely be coerced up (via subtyping) to match the setting of High on `Y`. More precisely, by rule [T-TRANS] (see Section 4.5), to type `A` $\overset{0}{\rightsquigarrow}$ `X.transfer(Y,1)` as a $\mathsf{cmd}(H)$ under some Σ, Γ and Δ, we must infer

$$\Sigma; \Gamma; \Delta' \vdash \texttt{X.transfer(Y,1)\$0} : \mathsf{cmd}(H)$$

where $\Delta' = \texttt{this} : \mathsf{var}(\Gamma(\texttt{A}))$. By rule [T-CALL], this amounts to have, among other things, that

$$\Sigma; \Gamma; \Delta' \vdash \texttt{X} : \texttt{IBank}_H$$

[8]It also involves the presence of a 'fallback function', which is a special feature of Solidity. It is a parameterless function that is implicitly invoked by `send()`, thus allowing the recipient to execute code upon reception of a currency transfer. This feature is not present in TinySol, yet we can achieve a similar effect by simply allowing the mandatory `send()` method to have an arbitrary method body, rather than just `skip`. This has no effect on the type system and associated proofs, since the `send()` method is treated as any other method therein. Hence, this situation is in principle the same as if the sender had invoked some other method than `send()`, similarly to the example in Figure 1.

[9]TinySol does not have a 'mapping' type such as in Solidity, so the setup here is limited to a single user.

If, as we assume, X has been declared as $\texttt{IBank}_L$, by rule [T-DEC-C] we have that $\Gamma(\texttt{X}) = \texttt{IBank}_L$; thus, by rule [T-VAL], we can infer that $\Sigma;\Gamma;\Delta' \vdash \texttt{X} : \texttt{IBank}_L$ and, by rule [SUB-TYPE], obtain the desired $\Sigma;\Gamma;\Delta' \vdash \texttt{X} : \texttt{IBank}_H$. By the other conditions of rule [T-CALL], we can conclude that A can have any security level $s_1 \sqsubseteq H$. In contrast, the balance fields of both A and X must be H (because of the side condition $H \sqsubseteq s_2, s_3$) and the method transfer must have type $\mathsf{proc}(\texttt{IBank}_H, \mathsf{int}_s) : H$, for some s, since Y has been declared as $\texttt{IBank}_H$.

Finally, we provide some typing insights on delegate calls. Consider the pointer to implementation pattern given in Figure 3. Assume that contract Y has been assigned some interface I and security level s_1; hence, we have to infer that

$$\Sigma;\Gamma;\Delta \vdash \texttt{contract Y : I}_{s_1}\ \{DF\ DM\}$$

where DF and DM are the field and method declarations for Y. In particular, this requires us to infer, by rule [T-DEC-C], that

$$\Sigma;\Gamma;\Delta \vdash_{\texttt{Y}} \texttt{f}_i\texttt{(}\tilde{x}\texttt{) \{dcall impl.f}_i\texttt{(}\tilde{x}\texttt{)\}}$$

Now, [T-DEC-M] must be used, and its premises give us that

$$\Gamma(\texttt{Y}) = (\texttt{I}, s_1) \qquad \Gamma(\texttt{I})(\texttt{balance}) = \mathsf{var}(\mathsf{int}, s_2)$$
$$\Gamma(\texttt{I})(\texttt{f}_i) = \mathsf{proc}(\widetilde{B_i, s_i'}) : s \qquad \Sigma;\Gamma;\Delta' \vdash \texttt{dcall impl.f}_i\texttt{(}\tilde{x}\texttt{)} : \mathsf{cmd}(s)$$

where $\Delta' = \texttt{this} : \mathsf{var}(\texttt{I}, s_1), \texttt{value} : \mathsf{var}(\mathsf{int}, s_2), \texttt{sender} : \mathsf{var}(I^\top, s_\top), \tilde{x} : (\widetilde{B_i, s_i'})$. Then, [T-DCALL] is needed to infer the last judgement; from its premises, we have that

$$\Sigma;\Gamma;\Delta' \vdash \texttt{impl} : (\texttt{I}', s) \qquad \Sigma \vdash (\texttt{I}, s_1) <: (\texttt{I}', s)$$

To infer the last judgement, it must be that $s_1 \sqsubseteq s$ and that there must exist a $j \geq 0$ such that $\overbrace{\Sigma(\ldots\Sigma(}^{j}\texttt{I}\,)\ldots) = \texttt{I}'$ (because of the combination of j applications of rule [SUB-NAME] and $j-1$ applications of rule [SUB-TRANS]). Finally,

$$\Gamma(\texttt{I}')(\texttt{f}) = \mathsf{proc}(\widetilde{B_i, s_i'}) : s \qquad \Sigma;\Gamma;\Delta' \vdash \tilde{x} : (\widetilde{B_i, s_i'})$$
$$\forall q \in (\mathrm{dom}\,(\Gamma(I')) \cap (\mathsf{FNames} \cup \{\texttt{balance}\})).\ \Gamma(\texttt{I})(q) = \Gamma(\texttt{I'})(q)$$

With this in mind, we can complete the typing of the pointer to implementation pattern with the definitions provided in Figure 20. In particular, we can set $s_1 = s_\bot = L$ and $s = s_\top = H$, assuming a two-level security lattice. If this were an ordinary call, the side condition $s \sqsubseteq s_2, s_3$ on the rule [T-CALL] would force both balance fields to be at level H, even though there is no credit movement between the caller and the callee. However, this side condition is not present in rule [T-DCALL], so we can also accept, e.g., $s_2 = s_3 = L$, because there is no information flow to and from the balance fields in the case of a delegate call.

6 Related Work

In light of the visibility and immutability of smart contracts, which makes it hard to correct errors once they are deployed in the wild, it is not surprising that there has been a substantial research effort within the formal methods community on developing formal techniques to prove safety properties of those programs—see, for instance, [50] for a survey. The literature on this topic is already huge and the whole gamut of techniques from the field of verification and validation has been adapted to the smart-contract setting. For example, this includes contributions employing frameworks based on finite-state machines to design and synthesise Ethereum smart contracts [35], a variety of static analysis techniques and accompanying tools, such as those presented in [19, 31, 44, 51], and deductive verification [12, 13, 38], amongst others. The Dafny-based approach reported in [13] is

```
contract Y : I_L {
  field balance := 10;
  field owner := A;
  field impl := X;
  send() { skip }
  Update(addr) {
    if sender = owner
      then impl := addr
      else skip
  }
  f_1(x̃) { dcall impl.f_1(x̃) }
  ⋮
  f_h(x̃) { dcall impl.f_h(x̃) }
}
```

```
interface I : I′ {
  balance : var(int, s_2)
  impl : var(I′_H)
  send : proc():L
  Update : proc(I′_H):H
  f_1 : proc(B̃_1, s′_1):H
  ⋮
  f_h : proc(B̃_h, s′_h):H
}
```

```
contract X : I′_H {
  field balance := 20;
  send() { skip }
  f_1(x̃) { S_1 }
  ⋮
  f_h(x̃) { S_h }
}
```

```
interface I′ : I^⊤ {
  balance : var(int, s_3)
  send : proc():L
  f_1 : proc(B̃_1, s′_1):H
  ⋮
  f_h : proc(B̃_h, s′_h):H
}
```

Fig. 20. A possible realization for the *pointer to implementation* pattern.

able to model arbitrary reentrancy in a setting with the 'gas mechanism', whereas [8] presents a way to analyse safety properties of smart contracts exhibiting reentrancy in a gas-free setting.

The study in [29] is close in spirit to ours in that it proposes a type system to ensure the absence of information flows that violate integrity policies in Solidity smart contracts. That work also presents a type verifier and its prototype implementation within the K-framework [41], which is then applied to analyse more than 100 smart contracts. However, their technique has not been related to call integrity, which, by contrast, is the focus of our work. Thus, our contribution in the present article complements this work and serves to further highlight the utility and applicability of secure-flow types in the smart-contract setting. However, there are also clear differences between this aforementioned work and the present one. Most notably, our type system uses a more refined subtyping relation, which also handles subtyping of method and address types, whereas subtyping is not defined for the former in [29], and the latter is not given a type altogether. This gives us a more fine-grained control over the information flow, since it allows us to assign different security levels to a contract and its members. For example, a High contract might have certain Low methods, which hence would not be callable from another High contract, whereas High methods would. This is in line with standard object-oriented principles, e.g., Java-style visibility modifiers.

Another approach to using a type system to ensure smart-contract safety in a Solidity-like language is presented by Crafa et al. in [16]. This work is indeed related to ours in that both are based on well-known typing principles from object-oriented languages, especially subtyping for contract/address types and the inclusion of a 'default' supertype for all contracts, similar to our $I^\top$. However, the aim of [16] is rather different from ours, in that the type system offered in that paper seeks to prevent *runtime errors* that do not stem from a negative account balance, e.g., those resulting from attempts to access nonexistent members of a contract. Incidentally, such runtime

errors would *also* be prevented by our type system (rules [T-CALL] and [T-ASSF] in particular), due to our use of 'interfaces' as address types, if the converse check (ensuring every declared type in an interface has an implementation) were also performed. However, our focus has been on checking the currency flow, rather than preventing runtime errors of this kind.

The aforementioned paper [16] introduced FS. Like TinySol, FS is a calculus that formalises the core features of Solidity and, as mentioned above, it supports the static analysis of safety properties of smart contracts via type systems. Therefore, the developments in the present article might conceivably have been carried out in FS instead of TinySol. Our rationale for using TinySol is that it provided a very simple language that was sufficient to express the property of call integrity, thus allowing us to focus on the core of this property. Of course, 'simplicity' is a subjective criterion and the choice of one language instead of another is often a matter of preference and convenience. To our mind, TinySol is slightly simpler than FS, which includes functionalities such as fallback functions and different kinds of exceptions. Moreover, the big-step semantics of TinySol provided was more convenient for the development of our type system than the small-step semantics given for FS. Furthermore, unlike FS, TinySol also formalises the semantics of blockchains. Having said so, TinySol and FS are quite similar and it would be interesting to study their similarities in more detail. To this end, in future work, we intend to carry out a formal comparison of these two core languages and to see which adaptations to our type system are needed when formulated for FS. In particular, we note that FS handles the possibility of an explicit type conversion (type cast) of `address` to `address payable` by augmenting the `address` type with type information about the contract to which it refers. This distinction is not present in our version of TinySol, as we require all contracts and accounts to have a default `send()` function, so all addresses are in this sense 'payable'. However, our type system does not depend on the presence of a `send()` function, so this difference is not important here.

The type system we present in this article builds on the seminal work by Volpano et al. on ensuring secure information flow by typing [53]. The basic types we use for the core fragment of TinySol and their meaning stem from that reference. However, applying the ideas of Volpano, Irvine and Smith to the setting of a smart contract language like TinySol involved handling constructs such as contract and method declarations, the 'magic variables', and the specific call styles supported by smart contract languages. Doing so created new challenges in the analysis of information flow in programs that, to the best of our knowledge, had not been handled in the literature before.[10]

The work on type systems for information-flow control by Volpano et al. [53] has been very influential in the field of language-based security and the literature on that topic is vast; see, for example, the survey given in [43] and the references therein, and the article [40] for a comparative study of the expressiveness of coarse- and fine-grained approaches in tracking information-flow dependencies.

To our mind, one of the reasons for the impact of the type-based approach to information-flow control pioneered by Volpano, Irvine and Smith is that it is flexible and can therefore be applied in a variety of settings. Indeed, apart from their applications to general-purpose programming languages [9, 36, 39, 42], type systems and related techniques for information-flow control have also been defined to support the safe integration of third-party code in applications while protecting the confidentiality of information [24], in the setting of session types [11] and the π-calculus [28, 32], as well as for languages for web applications, see for instance [27] and the references therein, and for programmable networks, such as P4 [25]. Our work contributes to that line of research

[10]We remark that in [53, Section 7.1] Volpano et al. mentioned that they extended their core language with procedures, akin to those in Ada 83, with at most three parameters. The technical treatment of that extension can be found in [52]. However, dealing with method calls in smart contract languages requires one to handle their specific features influencing information flow in programs.

by employing the ideas underlying type systems for information-flow control to smart contract languages.

7 Conclusion and Future Work

In this article we studied two security properties, namely call integrity and noninterference, in the setting of TinySol, a minimal calculus for Solidity smart contracts. To this end, we rephrased the syntax of TinySol to emphasise its object-oriented flavour, gave a new big-step operational semantics for that language and used it to define call integrity and noninterference. Those two properties have some similarities in their definition, in that they both require that some part of a program is not influenced by the other part. However, we showed that the two properties are actually incomparable. Nevertheless, we provided a type system for noninterference and showed that well-typed programs *also* satisfy call integrity. Hence, programs that are accepted by our type system lie at the intersection of call integrity and noninterference. A challenging development of our work would be to prove whether the type system exactly characterises the intersection of these two properties, or to find another characterisation of this set of programs.

There is a publicly available implementation of an interpreter for TinySol, together with a type checker based on the type system described in the present article. The code for this implementation is available in [21]. The repository also contains a small sample of typable and untypable contracts drawn from the conference version of the present article [1]. A direction for future work is to apply our static analysis methodology (and the associated implementation) to concrete case studies, to better understand the benefits of using a completely static proof technique for call integrity. This may also include expanding the collection of examples with TinySol versions of actual contracts, to be able to carry out a larger-scale testing of the type checker, similar to the experiments in [29]. To do so, it would be useful to consider the extension of TinySol with a 'gas mechanism' [2], allowing one to prove the termination of transactions and to compute their computational cost.

However, many actual contracts also rely on a feature that we have not included in the present formulation of TinySol, namely the *fallback function*. This construct is a peculiar feature of the Solidity smart-contract language, which was the source of a spectacular security breach (the so-called *Parity Wallet Hack*, described in e.g., [29]). A fallback function is a special, parameterless method, which is executed in case a method call *fails*, because the callee contract does not contain a method by the called name. With fallback functions, in such a case, control would *still* be transferred to the callee contract, and the callee can then use the fallback function to perform some error handling, or even to forward the call to some other method via a delegate call. This feature is also used to implement a form of dynamic dispatch of method calls in Solidity. Unfortunately, the fallback function construct is untypable by ordinary syntactic means, such as the type system presented in the present article. It admits a form of introspection into the language, because the body of a fallback function has access to information *about* the failed method call, such as the name of the non-existing method, and the list of actual parameters. This information depends on how the method was called (i.e., on the form of the failed method call), and it is therefore only available at runtime. Hence, it cannot be given a static type, which is necessary for a static type check. On the other hand, the present type system already prohibits calls to non-existing methods because of our interface types for contracts, thereby rendering fallback functions useless in well-typed programs. This is doubly problematic from a practical perspective, since the fallback function *is* part of the Solidity language. In [34, Chapter 8], a method based on a *semantic approach* to type soundness (cf. [3, 10, 49]) is developed to allow type safety to be recovered for TinySol contracts containing fallback functions, without disallowing usage of the construct altogether.

A potential limitation of the approach presented in this article is that the entire blockchain must be checked to show call integrity of a contract. Indeed, since a typing derivation can fail at the

```
contract X : I_L {
  field a := 42;
  field b := 0;
  write(x) {
   this.b := x }
  ...
}
```

```
interface I : I^⊤ {
  a     : var(int_H)
  b     : var(int_L)
  write : proc(int_L):L
  ...
}
```

```
contract Y : I'_H {
  f() {
    call X.write(X.a)$0
  }
  ...
}
```

Fig. 21. An illustration of the problem of mixing well-typed and ill-typed code.

call-site and the call-site of a method can be a transaction, transactions must be well-typed too. In passing, we note that this kind of problem is also present in [47, 53] (and, in general, in many works on type systems for security), where the whole code needs to be typed in order to obtain the desired guarantees. The type system as presented in this article is *not* compositional, in the sense that we cannot combine a piece of typed code and a piece of untyped code (or even typed, but under different typing environments) and be sure that the resulting program is still typable. This can be illustrated with a simple example: Consider the code in Figure 21. Contract `X` is well-typed with interface I; it has two fields, `a` and `b`, which have security levels H and L, respectively. It also has the method `write(x)`, which overwrites the value of `b` with its received parameter; hence the method has type $\mathsf{proc}(\mathsf{int}_L) : L$, because the level of the parameter `x` must be L. However, if untyped contracts are permitted to be added, then nothing would prevent an attacker from creating a contract such as `Y`, containing a method such as `f`, which simply calls `X.write` with the High value `X.a` as parameter. This creates an illegal flow of a High value into a Low field. An attacker can obviously not be expected to only write well-typed code, but if ill-typed code is permitted, then no guarantees can be given for even the well-typed part of the code. At the moment, one possible way to handle a scenario like this is to defer some type checks at runtime, as prescribed, for example, in the *gradual typing* approach [14, 46].

A related issue concerns dynamic contract creation. In the present setting, we have assumed that all contracts are created at the Genesis event, but on a real blockchain new contracts may of course be added at any moment. This can be accommodated in TinySol by changing the syntax of blockchains to allow contract declarations to appear within transactions (hence, a transaction alternates sequences of method invocations and contract creations) and their execution dynamically modifies both the state env_S and the method table env_T. However, as we argued above, the new contracts must be well-typed as part of the transaction (of course, relative to some type environments Σ' and Γ', which extend the previous environments Σ and Γ with the new declarations). The resulting setting is still a 'closed-world' one, where all the blockchain is typed. Therefore, this way of dynamically creating new contracts cannot deal with an 'open-world' scenario, where parts of the blockchain are totally unknown. In fact, an important avenue for future work, and one we intend to pursue, is to explore whether, and to what extent, other typing disciplines (such as gradual typing or the semantic approach mentioned above) can be employed to mitigate this problem.

Acknowledgments

We thank the anonymous reviewers for their detailed comments that helped us improve our article. Moreover, Luca Aceto and Stian Lybech thank Mohammad Hamdaqa for sharing his expertise with them during extensive discussions on safety properties for smart contracts, which helped shape the research agenda for the work reported in this article.

References

[1] Luca Aceto, Daniele Gorla, and Stian Lybech. 2024. A sound type system for secure currency flow. In *38th European Conference on Object-Oriented Programming (ECOOP '24)*. Jonathan Aldrich and Guido Salvaneschi (Eds.), Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 1–27. DOI: https://doi.org/10.4230/LIPIcs.ECOOP.2024.1. Retrieved from https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2024.1

[2] Luca Aceto, Daniele Gorla, Stian Lybech, and Mohammad Hamdaqa. 2025. Preventing out-of-gas exceptions by typing. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. REoCAS Colloquium in Honor of Rocco De Nicola*. Tiziana Margaria and Bernhard Steffen (Eds.), Springer Nature Switzerland, Cham, 409–426.

[3] Amal Jamil Ahmed. 2004. *Semantics of Types for Mutable State*. PhD thesis, Princeton University. Retrieved from http://www.ccs.neu.edu/home/amal/ahmedsthesis.pdf

[4] Aslan Askarov and Andrew C. Myers. 2011. Attacker control and impact for confidentiality and integrity. *Log. Methods Comput. Sci.* 7, 3 (2011), 1–33. DOI: https://doi.org/10.2168/LMCS-7(3:17)2011

[5] Aslan Askarov and Andrei Sabelfeld. 2007. Gradual release: Unifying declassification, encryption and key release policies. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, 207–221. DOI: https://doi.org/10.1109/SP.2007.22

[6] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. 2017. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust (POST '17)*. M. Maffei and M. Ryan, M. (Eds.), Lecture Notes in Computer Science, Vol. 10204, Springer, Berlin, Heidelberg. DOI: https://doi.org/10.1007/978-3-662-54455-6_8

[7] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. 2019. A minimal core calculus for solidity contracts. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*. Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro (Eds.), Springer International Publishing, Cham, 233–243. DOI: https://doi.org/10.1007/978-3-030-31500-9_15

[8] Christian Bräm, Marco Eilers, Peter Müller, Robin Sierra, and Alexander J. Summers. 2021. Rich specifications for Ethereum smart contract verification. *Proc. ACM Program. Lang.* 5, OOPSLA (2021), 1–30. DOI: https://doi.org/10.1145/3485523

[9] Pablo Buiras, Dimitrios Vytiniotis, and Alejandro Russo. 2015. HLIO: Mixing static and dynamic typing for information-flow control in Haskell. In *20th ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. Kathleen Fisher and John H. Reppy (Eds.), ACM, 289–301. DOI: https://doi.org/10.1145/2784731.2784758

[10] Luís Caires. 2007. Logical semantics of types for concurrency. In *Algebra and Coalgebra in Computer Science*. Till Mossakowski, Ugo Montanari, and Magne Haveraaen (Eds.), Springer, Berlin, Heidelberg, 16–35. DOI: https://doi.org/10.1007/978-3-540-73859-6_2

[11] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. 2010. Session types for access and information flow control. In *21st International Conference on Concurrency Theory (CONCUR '10)*. Paul Gastin and François Laroussinie (Eds.), Lecture Notes in Computer Science, Vol. 6269, Springer, 237–252. DOI: https://doi.org/10.1007/978-3-642-15375-4_17

[12] Franck Cassez, Joanne Fuller, and Aditya Asgaonkar. 2022. Formal verification of the Ethereum 2.0 Beacon Chain. In *28th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science, Vol. 13243, Springer, 167–182. DOI: https://doi.org/10.1007/978-3-030-99524-9_9

[13] Franck Cassez, Joanne Fuller, and Horacio Mijail Anton Quiles. 2022. Deductive verification of smart contracts with Dafny. In *27th International Conference on Formal Methods for Industrial Critical Systems*. J. F. Groote and M. Huisman (Eds.), Lecture Notes in Computer Science, Vol. 13487, Springer, 50–66. DOI: https://doi.org/10.1007/978-3-031-15008-1_5

[14] Giuseppe Castagna, Victor Lanvin, Tommaso Petrucciani, and Jeremy G. Siek. 2019. Gradual typing: A new perspective. *Proc. ACM Program. Lang.* 3, POPL Article 16 (2019), 1–32. DOI: https://doi.org/10.1145/3290329

[15] Michael R. Clarkson and Fred B. Schneider. 2010. Hyperproperties. *J. Comput. Secur.* 18, 6 (2010), 1157–1210. DOI: https://doi.org/10.3233/JCS-2009-0393

[16] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. 2019. Is solidity solid enough? In *Financial Cryptography Workshops*.

[17] DAO. 2016. The DAO smart contract. Retrieved from http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code

[18] Thomas D. Dickerson, Paul Gazzillo, Maurice Herlihy, Vikram Saraph, and Eric Koskinen. 2018. Proof-carrying smart contracts. In *Financial Cryptography Workshops*.

[19] Josselin Feist, Gustavo Grieco, and Alex Groce. Slither: A static analysis framework for smart contracts. In *2nd International Workshop on Emerging Trends in Software Engineering for Blockchain*. IEEE/ACM, 8–15. DOI: https://doi.org/10.1109/WETSEB.2019.00008

[20] Ethereum Foundation. 2022. Solidity Documentation. Retrieved January 15, 2024 from https://docs.soliditylang.org/

[21] Arnór Friðriksson. 2024. Interpreter and Type Checker for TinySol. Retrieved March 3, 2024 from https://github.com/Zepeacedust/TinySol-Type-checker
[22] Thomas Genet, Thomas P. Jensen, and Justine Sauvage. 2020. Termination of Ethereum's smart contracts. In *17th International Joint Conference on e-Business and Telecommunications (SECRYPT)*, Vol. 2, ScitePress, 39–51. DOI: https://doi.org/10.5220/0009564100390051
[23] J. A. Goguen and J. Meseguer. 1982. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, 11–20. DOI: https://doi.org/10.1109/SP.1982.10014
[24] Simon Oddershede Gregersen, Søren Eller Thomsen, and Aslan Askarov. 2019. A dependently typed library for static information-flow control in Idris. In *8th International Conference on Principles of Security and Trust (POST '19)*. Flemming Nielson and David Sands (Eds.), Lecture Notes in Computer Science, Vol. 11426, Springer, 51–75. DOI: https://doi.org/10.1007/978-3-030-17138-4_3
[25] Karuna Grewal, Loris D'Antoni, and Justin Hsu. 2022. P4BID: information flow control in P4. In *43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. Ranjit Jhala and Isil Dillig (Eds.), ACM, 46–60. DOI: https://doi.org/10.1145/3519939.3523717
[26] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. 2018. A semantic framework for the security analysis of Ethereum smart contracts. In *Principles of Security and Trust*. Lujo Bauer and Ralf Küsters (Eds.), Springer International Publishing, Cham, 243–269.
[27] Daniel Hedin, Arnar Birgisson, Luciano Bello, and Andrei Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Symposium on Applied Computing (SAC '14)*. Yookun Cho, Sung Y. Shin, Sang-Wook Kim, Chih-Cheng Hung, and Jiman Hong (Eds.), ACM, 1663–1671. DOI: https://doi.org/10.1145/2554850.2554909
[28] Kohei Honda and Nobuko Yoshida. 2007. A uniform type structure for secure information flow. *ACM Trans. Program. Lang. Syst.* 29, 6 (2007), 31. DOI: https://doi.org/10.1145/1286821.1286822
[29] Xinwen Hu, Yi Zhuang, Shangwei Lin, Fuyuan Zhang, Shuanglong Kan, and Zining Cao. A security type verifier for smart contracts. *Comput. Secur.* 108 (2021), 102343. DOI: https://doi.org/10.1016/j.cose.2021.102343
[30] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.* 23, 3 (2001), 396–450. DOI: https://doi.org/10.1145/503502.503505
[31] Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. 2018. ZEUS: analyzing safety of smart contracts. In *25th Annual Network and Distributed System Security Symposium*. The Internet Society. Retrieved from https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_09-1_Kalra_paper.pdf
[32] Naoki Kobayashi. 2005. Type-based information flow analysis for the pi-calculus. *Acta Informatica* 42, 4–5 (2005), 291–347. DOI: https://doi.org/10.1007/s00236-005-0179-x
[33] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. 2016. Making smart contracts smarter. In *SIGSAC Conf. on Computer and Communications Security*. ACM, 254–269. DOI: https://doi.org/10.1145/2976749.2978309
[34] Stian Lybech. 2025. *A Type-Theoretic Approach to Smart-Contract Safety*. PhD thesis, Reykjavík University. Retrieved from https://iris.landsbokasafn.is/en/publications/a-type-theoretic-approach-to-smart-contract-safety
[35] Anastasia Mavridou and Aron Laszka. 2018. Designing secure Ethereum smart contracts: A finite state machine based approach. In *22nd Conference on Financial Cryptography and Data Security*. S. Meiklejohn and K. Sako (Eds.), Lecture Notes in Computer Science, Vol. 10957, Springer, 523–540. DOI: https://doi.org/10.1007/978-3-662-58387-6_28
[36] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. Jif 3.0: Java Information Flow. Retrieved from http://www.cs.cornell.edu/jif
[37] Hanne Riis Nielson and Flemming Nielson. 2007. *Semantics with Applications: An Appetizer*. Springer-Verlag London. DOI: https://doi.org/10.1007/978-1-84628-692-6
[38] Daejun Park, Yi Zhang, and Grigore Rosu. End-to-end formal verification of Ethereum 2.0 Deposit Smart Contract. In *32nd International Conference Computer Aided Verification (CAV), Part I*. Shuvendu K. Lahiri and Chao Wang (Eds.), Lecture Notes in Computer Science, Vol. 12224, Springer, 151–164. DOI: https://doi.org/10.1007/978-3-030-53288-8_8
[39] François Pottier and Vincent Simonet. 2003. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1 (2003), 117–158. DOI: https://doi.org/10.1145/596980.596983
[40] Vineet Rajani, Iulia Bastys, Willard Rafnsson, and Deepak Garg. 2017. Type systems for information flow control: The question of granularity. *ACM SIGLOG News* 4, 1 (2017), 6–21. DOI: https://doi.org/10.1145/3051528.3051531
[41] Grigore Rosu and Traian-Florin Serbanuta. 2010. An overview of the K semantic framework. *J. Log. Algebraic Methods Program* 79, 6 (2010), 397–434. DOI: https://doi.org/10.1016/j.jlap.2010.03.012
[42] Alejandro Russo. 2015. Functional pearl: Two can keep a secret, if one of them uses Haskell. In *20th ACM SIGPLAN International Conference on Functional Programming (ICFP '15)*. Kathleen Fisher and John H. Reppy (Eds.), ACM, 280–288. DOI: https://doi.org/10.1145/2784731.2784756
[43] Andrei Sabelfeld and Andrew C. Myers. 2003. Language-based information-flow security. *IEEE J. Sel. Areas Commun.* 21, 1 (2003), 5–19. DOI: https://doi.org/10.1109/JSAC.2002.806121

[44] Clara Schneidewind, Ilya Grishchenko, Markus Scherer, and Matteo Maffei. 2020. eThor: Practical and provably sound static analysis of ethereum smart contracts. In *SIGSAC Conference on Computer and Communications Security*. ACM, 621–640. DOI: https://doi.org/10.1145/3372297.3417250
[45] Pablo Lamela Seijas, Simon J. Thompson, and Darryl McAdams. 2016. Scripting smart contracts for distributed ledger technology. *IACR Cryptol. ePrint Arch.* (2016), 1156.
[46] Jeremy G. Siek and Walid Taha. 2007. Gradual typing for objects. In *European Conference on Object Oriented Programming (ECOOP)*. Erik Ernst (Ed.), Lecture Notes in Computer Science, Vol. 4609, Springer, 2–27, DOI: https://doi.org/10.1007/978-3-540-73589-2_2
[47] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 355–364. ACM.
[48] smlxl inc. 2023. An Ethereum virtual machine opcodes interactive reference, Retrieved from https://evm.codes/
[49] Amin Timany, Robbert Krebbers, Derek Dreyer, and Lars Birkedal. A logical approach to type soundness. *J. ACM* 71, 6, Article 40 (Nov. 2024), 1–75. DOI: https://doi.org/10.1145/3676954
[50] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. A survey of smart contract formal specification and verification. *ACM Computing Surveys* 54, 7, Article 148 (2020), 1–38. DOI: https://doi.org/10.1145/3464421
[51] Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. 2018. Securify: Practical security analysis of smart contracts. In *SIGSAC Conference on Computer and Communications Security*. ACM, 67–82. DOI: https://doi.org/10.1145/3243734.3243780
[52] Dennis M. Volpano and Geoffrey Smith. 1997. A type-based approach to program security. In *7th International Joint Conference on Theory and Practice of Software Development CAAP/FASE (TAPSOFT '97)*. Michel Bidoit and Max Dauchet (Eds.), Lecture Notes in Computer Science, Vol. 1214, Springer, 607–621, 1997. DOI: https://doi.org/10.1007/BFb0030629
[53] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A sound type system for secure flow analysis. *J. Comput. Security* 4, 2/3 (1996), 167–188. DOI: https://doi.org/10.3233/JCS-1996-42-304
[54] Renlord Yang, Toby Murray, Paul Rimba, and Udaya Parampalli. 2019. Empirically analyzing Ethereum's gas mechanism. In *IEEE European Symposium on Security and Privacy Workshops*. IEEE, 310–319. DOI: https://doi.org/10.1109/EuroSPW.2019.00041

Appendices

A Proof of Theorem 1

Proof. By induction on the derivation of $\Sigma; \Gamma; \Delta \vdash e : B_s$. There are two base cases:

—If [T-VAL] was used to conclude $\Gamma \vdash e : B_s$, then $e = v$. The result is immediate, since the value does not depend on env_{SV}.
—If [T-VAR] was used, then $e = k$ and, from the side condition of the rule, we have that $\Delta(k) = \mathsf{var}(B_s)$. Moreover, by the premise of [VAR], $\mathsf{env}_V(k) = v$. By assumption, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_V$ which was concluded by [T-ENV-V], and from the premise of this rule we have that $\Sigma; \Gamma; \Delta \vdash v : B_s$.

For the inductive case, we distinguish the last rule used in the inference; we have the following three cases:

—If [T-FIELD] was used, then $e = e'.q$. From the premise and side condition of that rule, we know that $\Sigma; \Gamma; \Delta \vdash e' : I_s$ and $\Gamma(I)(q) = \mathsf{var}(B_s)$. By assumption, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_{SV}$. Therefore, if $\mathsf{env}_{SV} \vdash e' \to_{\mathrm{e}} X$, then by the induction hypothesis $\Sigma; \Gamma; \Delta \vdash X : I_s$.
Now, $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_S$ was concluded by [T-ENV-S]; by considering the pair (X, env_F), we know from the premise that $\Sigma; \Gamma; \Delta \vdash_X \mathsf{env}_F$. This, in turn, was concluded by [T-ENV-F]; by considering the pair (q, v), we know from the premise of that rule that $\Sigma; \Gamma; \Delta \vdash v : B_s$.
—If [T-OP] was used, then $e = \mathrm{op}(e_1, \ldots, e_h)$. From the premise, we know that each of the arguments e_i are typable as $\Sigma; \Gamma; \Delta \vdash e_i : (B_i, s)$. Thus, if $\mathsf{env}_{SV} \vdash e_i \to_{\mathrm{e}} v_i$ for each $i \in \{1, \ldots, h\}$, we get by h applications of the induction hypothesis that $\Sigma; \Gamma; \Delta \vdash v_i : (B_i, s)$. Now, if $\mathrm{op}(v_1, \ldots, v_h) \to_{\mathrm{op}} v$, then by the safety requirement for operations (Definition 11), we have that $\textsc{TypeOf}_\Gamma(v) = (B, s_\perp)$. Thus, $\Sigma; \Gamma; \Delta \vdash v : B_s$ can be concluded as well by subtyping, since by definition $s_\perp \sqsubseteq s$ for any s.

—If [T-ESUB] was used, then we know from the premise that $\Sigma; \Gamma; \Delta \vdash e : (B', s')$ and $\Sigma \vdash (B', s') <: (B, s)$. Thus, if $\mathsf{env}_{SV} \vdash e \to_{\mathrm{e}} v$, then by the induction hypothesis we have that $\Sigma; \Gamma; \Delta \vdash v : (B', s')$. Then $\Sigma; \Gamma; \Delta \vdash v : (B, s)$ can be concluded by [T-ESUB]. □

B Proof of Theorem 2

Proof. By induction on the derivation of $\Sigma; \Gamma; \Delta \vdash e : B_s$. There are two base cases:

—If [T-VAL] was used to conclude $\Gamma \vdash e : B_s$, then $e = v$. The result is immediate, since the value does not depend on env_{SV}.
—If [T-VAR] was used, then $e = k$ and, from the side condition of the rule, we have that $\Delta(k) = \mathsf{var}(B_s)$. By assumption, $\Gamma; \Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$; so, by [EQ-ENV$_V$], $\mathsf{env}^1_V(k) = \mathsf{env}^2_V(k)$. Thus, we can conclude by [VAR].

For the inductive case, we distinguish the last rule used in the inference; we have the following three cases:

—If [T-FIELD] was used, then $e = e'.q$; moreover, from the premise and side condition of that rule, we know that $\Sigma; \Gamma; \Delta \vdash e' : I_s$ and $\Gamma(I)(q) = \mathsf{var}(B_s)$. By assumption, $\Gamma; \Delta \vdash \mathsf{env}^1_{SV} =_s \mathsf{env}^2_{SV}$; therefore, by the induction hypothesis, $\mathsf{env}^1_{SV} \vdash e' \to_{\mathrm{e}} X$ and $\mathsf{env}^2_{SV} \vdash e' \to_{\mathrm{e}} X$. Then, because $\mathsf{env}^1_S =_s \mathsf{env}^2_S$, we also have that $\mathsf{env}^1_S(X)(q) = \mathsf{env}^2_S(X)(q)$ and we can conclude by [FIELD].
—If [T-OP] was used, then $e = \mathrm{op}(e_1, \ldots, e_h)$. From the premise, we know that each of the arguments e_i are typable as $\Sigma; \Gamma; \Delta \vdash e_i : (B_i, s)$; so, by h applications of the induction hypothesis, we get that $\mathsf{env}^1_{SV} \vdash e_i \to_{\mathrm{e}} v_i$ and $\mathsf{env}^2_{SV} \vdash e_i \to_{\mathrm{e}} v_i$. We can conclude by rule [OP].
—If [T-ESUB] was used, then we know from the premise that $\Sigma; \Gamma; \Delta \vdash e : B'_{s'}$ and $\Sigma \vdash (B', s') <: (B, s)$. By rule [SUB-TYPE], we know that $s' \sqsubseteq s$; hence, by Lemma 1, $\Gamma; \Delta \vdash \mathsf{env}^1_{SV} =_{s'} \mathsf{env}^2_{SV}$ also holds. The claim holds because of the induction hypothesis. □

C Proof of Lemma 5

Proof. Notationally, let $\Delta_1 = \Delta, \mathtt{this} : \mathsf{var}(I, s_1), \mathtt{value} : \mathsf{var}(\mathsf{int}, s_2)$ and $\Delta_2 = \Delta, \mathtt{this} : \mathsf{var}(I', s'_1), \mathtt{value} : \mathsf{var}(\mathsf{int}, s'_2)$. The proof is then by induction on the derivation of $\Sigma; \Gamma; \Delta_1 \vdash e : B_s$. There are two base cases:

—If [T-VAL] was used, then $e = v$. The result is immediate, since the typing of a value does not depend on any Δ-environment.
—If [T-VAR] was used, then $e = k$; from the side condition of the rule, we have that $\Delta_1(k) = \mathsf{var}(B_s)$. There are now two possibilities:
  –If $k \neq \mathtt{this}$ and $k \neq \mathtt{value}$, then the result is immediate, since $\Delta_2(k) = \mathsf{var}(B_s)$ as well.
  –Otherwise, we have that $\Delta_1(\mathtt{this}) = (I, s_1)$ and $\Delta_2(\mathtt{this}) = (I', s'_1)$, or $\Delta_1(\mathtt{value}) = (\mathsf{int}, s_2)$ and $\Delta_2(\mathtt{value}) = (\mathsf{int}, s'_2)$. By assumption, $\Sigma \vdash (I', s'_1) <: (I, s_1)$ and $s'_2 \sqsubseteq s_2$; so, by [SUB-TYPE], we also have that $\Sigma \vdash (\mathsf{int}, s'_2) <: (\mathsf{int}, s_2)$. Therefore, $\Sigma; \Gamma; \Delta_2 \vdash \mathtt{this} : (I, s_1)$ or $\Sigma; \Gamma; \Delta_2 \vdash \mathtt{value} : (\mathsf{int}, s_2)$ can be concluded as well, by [T-ESUB].

For the inductive case, we distinguish the last rule used in the inference; we have the following three cases:

—If [T-FIELD] was used, then $e = e'.q$. Moreover, from the premise and side condition of that rule, we know that $\Sigma; \Gamma; \Delta_1 \vdash e' : I_s$ and $\Gamma(I)(q) = \mathsf{var}(B_s)$.

If $e' \neq \texttt{this}$, then, by the induction hypothesis, we can also conclude $\Sigma; \Gamma; \Delta_2 \vdash e' : I_s$ and conclude by rule [T-FIELD].
If $e' = \texttt{this}$, then $\Sigma; \Gamma; \Delta_1 \vdash \texttt{this} : (I, s_1)$ and $\Sigma; \Gamma; \Delta_2 \vdash \texttt{this} : (I', s_1')$, where $\Sigma \vdash (I', s_1') <: (I, s_1)$. Now by assumption, $\Gamma(I)(q) = \Gamma(I')(q)$, for every $q \in \mathrm{dom}(\Gamma(I))$; therefore, $\Gamma(I')(q) = \mathsf{var}(B_s)$ as well. Then we conclude by [T-FIELD].

—If [T-OP] was used, then $e = \mathrm{op}(e_1, \ldots, e_h)$. From the premise, we know that each of the arguments e_i are typable as $\Sigma; \Gamma; \Delta_1 \vdash e_i : (B_i, s)$; so, by h applications of the induction hypothesis, we get that $\Sigma; \Gamma; \Delta_2 \vdash e_i : (B_i, s)$ can be concluded as well. Then we conclude by [T-OP].

—If [T-ESUB] was used, then we know from the premise that $\Sigma; \Gamma; \Delta_1 \vdash e : B'_{s'}$ and $\Sigma \vdash (B', s') <: (B, s)$. By the induction hypothesis, $\Sigma; \Gamma; \Delta_2 \vdash e : B'_{s'}$ can therefore also be concluded, and we can then conclude $\Sigma; \Gamma; \Delta_2 \vdash e : B_s$ by [T-ESUB]. □

D Proof of Lemma 6

Proof. Notationally, let $\Delta_1 = \Delta, \texttt{this} : \mathsf{var}(I, s_1), \texttt{value} : \mathsf{var}(\mathsf{int}, s_2)$ and $\Delta_2 = \Delta, \texttt{this} : \mathsf{var}(I', s_1'), \texttt{value} : \mathsf{var}(\mathsf{int}, s_2')$. The proof is then by induction on the derivation of $\Sigma; \Gamma; \Delta_1 \vdash S : \mathsf{cmd}(s)$. There are six base cases:

—If [T-SKIP] was used, the result is immediate since the Δ-environment is not used.

—If [T-THROW] was used, the result is immediate since the Δ-environment is not used.

—If [T-ASSV] was used, then $S = \texttt{x := e}$; from the premise and side condition, we know that $\Sigma; \Gamma; \Delta_1 \vdash e : B_s$ and $\Delta_1(x) = \mathsf{var}(B_s)$. By Lemma 5, we can therefore also conclude $\Sigma; \Gamma; \Delta_2 \vdash e : B_s$. Note that x cannot be a magic variable, since they cannot be assigned to directly. Thus, we also have that $\Delta_2(x) = \mathsf{var}(B_s)$, since the only difference between Δ_1 and Δ_2 are the types for this and value. We can therefore conclude $\Sigma; \Gamma; \Delta_2 \vdash x := e : \mathsf{cmd}(s)$ by [T-ASSV].

—If [T-ASSF] was used, then $S = \texttt{this}.p := e : \mathsf{cmd}(s)$; from the premise and side conditions, we know that $\Sigma; \Gamma; \Delta_1 \vdash e : B_s$, $\Gamma(I)(p) = \mathsf{var}(B_s)$, and $\Delta_1(\texttt{this}) = (I, s_1)$, for $s_1 \sqsubseteq s$. Moreover, $\Delta_2(\texttt{this}) = (I', s_1')$ and, by assumption, $\Sigma \vdash (I', s_1') <: (I, s_1)$, which implies that $s_1' \sqsubseteq s_1$ and, by transitivity, $s_1' \sqsubseteq s$. Furthermore, by assumption we have that $\Gamma(I)(q) = \Gamma(I')(q)$, for every $q \in \mathrm{dom}(\Gamma(I))$; hence, $\Gamma(I')(p) = \mathsf{var}(B_s)$ as well. Lastly, by Lemma 5, we know that $\Sigma; \Gamma; \Delta_2 \vdash e : B_s$; thus, we can conclude $\Sigma; \Gamma; \Delta_2 \vdash \texttt{this}.p := e : \mathsf{cmd}(s)$ by [T-ASSF].

—If [T-CALL] was used, then $S = e_1.f(\tilde{e})\$e_2$. There are three premises:

$$\Sigma; \Gamma; \Delta_1 \vdash e_1 : (I^Y, s) \qquad \Sigma; \Gamma; \Delta_1 \vdash e_2 : \mathsf{int}_s \qquad \Sigma; \Gamma; \Delta_1 \vdash \tilde{e} : \widetilde{B_{s'}}$$

and, by applying Lemma 5, we obtain

$$\Sigma; \Gamma; \Delta_2 \vdash e_1 : (I^Y, s) \qquad \Sigma; \Gamma; \Delta_2 \vdash e_2 : \mathsf{int}_s \qquad \Sigma; \Gamma; \Delta_2 \vdash \tilde{e} : \widetilde{B_{s'}}$$

We also have the side conditions $\Delta_1(\texttt{this}) = \mathsf{var}(I, s_1)$ and $s_1 \sqsubseteq s$. By assumption, $\Delta_2(\texttt{this}) = \mathsf{var}(I', s_1')$, where $\Sigma \vdash (I', s_1') <: (I, s_1)$, which implies that $s_1' \sqsubseteq s_1$ and, by transitivity, also $s_1' \sqsubseteq s$. We can then conclude $\Sigma; \Gamma; \Delta_2 \vdash e_1.f(\tilde{e})\$e_2 : \mathsf{cmd}(s)$ by [T-CALL].

—If [T-DCALL] was used, then $S = \Sigma; \Gamma; \Delta \vdash \texttt{dcall}\ e.f(\tilde{e}) : \mathsf{cmd}(s)$. There are three premises:

$$\Sigma; \Gamma; \Delta_1 \vdash e : (I^Y, s) \qquad \Sigma \vdash (I, s_1) <: (I^Y, s) \qquad \Sigma; \Gamma; \Delta_1 \vdash \tilde{e} : \widetilde{B_{s'}}$$

and, by applying Lemma 5, we obtain

$$\Sigma; \Gamma; \Delta_2 \vdash e : (I^Y, s) \qquad \Sigma; \Gamma; \Delta_2 \vdash \tilde{e} : \widetilde{B_{s'}}$$

Furthermore, we know by assumption that $\Sigma \vdash (I', s_1') <: (I, s_1)$; so, by transitivity, we also have that $\Sigma \vdash (I', s_1') <: (I^Y, s)$. Finally, by assumption, we know that

$$\forall q \in (\mathrm{dom}\,(\Gamma(I)) \cap (\mathsf{FNames} \cup \{\mathtt{balance}\})).\ \Gamma(I)(q) = \Gamma(I')(q)$$

From the side condition of rule [T-DCALL], we have that

$$\forall q \in (\mathrm{dom}\,(\Gamma(I^Y)) \cap (\mathsf{FNames} \cup \{\mathtt{balance}\})).\ \Gamma(I^Y)(q) = \Gamma(I)(q),$$

which implies that also

$$\forall q \in (\mathrm{dom}\,(\Gamma(I^Y)) \cap (\mathsf{FNames} \cup \{\mathtt{balance}\})).\ \Gamma(I^Y)(q) = \Gamma(I')(q).$$

Thus we can conclude $\Sigma; \Gamma; \Delta_2 \vdash \mathtt{dcall}\ e.f(\tilde{e}) : \mathsf{cmd}(s)$ by [T-DCALL].

For the inductive step, we distinguish the last rule used in the inference; we have the following five cases to consider:

- —if [T-DECV] was used, then $S = \mathsf{var}(B_{s'})\ x\ \mathtt{:=}\ e\ \mathtt{in}\ S$; from the premise, we know that $\Sigma; \Gamma; \Delta_1 \vdash e : B_{s'}$ and $\Sigma; \Gamma; \Delta_1, x : \mathsf{var}(B_{s'}) \vdash S : \mathsf{cmd}(s)$. By Lemma 5, we can obtain that $\Sigma; \Gamma; \Delta_2 \vdash e : B_{s'}$ and, by the induction hypothesis, that $\Sigma; \Gamma; \Delta_2, x : \mathsf{var}(B_{s'}) \vdash S : \mathsf{cmd}(s)$. Then, we can conclude $\Sigma; \Gamma; \Delta_2 \vdash \mathsf{var}(B_{s'})\ x\ \mathtt{:=}\ e\ \mathtt{in}\ S : \mathsf{cmd}(s)$ by [T-DECV].
- —If [T-WHILE] was used, then $S = \mathtt{while}\ e\ \mathtt{do}\ S$; from the premise, we know that $\Sigma; \Gamma; \Delta_1 \vdash e : \mathsf{bool}_s$ and $\Sigma; \Gamma; \Delta_1 \vdash S : \mathsf{cmd}(s)$. By Lemma 5, we obtain that $\Sigma; \Gamma; \Delta_2 \vdash e : \mathsf{bool}_s$ and, by the induction hypothesis, that $\Sigma; \Gamma; \Delta_2 \vdash S : \mathsf{cmd}(s)$. We can then conclude $\Sigma; \Gamma; \Delta_2 \vdash \mathtt{while}\ e\ \mathtt{do}\ S$ by [T-WHILE].
- —If [T-IF] was used, then $S = \mathtt{if}\ e\ \mathtt{then}\ S_\mathsf{T}\ \mathtt{else}\ S_\mathsf{F}$; from the premise, we know that $\Sigma; \Gamma; \Delta_1 \vdash e : \mathsf{bool}_s$, $\Sigma; \Gamma; \Delta_1 \vdash S_\mathsf{T} : \mathsf{cmd}(s)$, and $\Sigma; \Gamma; \Delta_1 \vdash S_\mathsf{F} : \mathsf{cmd}(s)$. By Lemma 5, we obtain that $\Sigma; \Gamma; \Delta_2 \vdash e : \mathsf{bool}_s$ and, by two applications of the induction hypothesis, that $\Sigma; \Gamma; \Delta_2 \vdash S_\mathsf{T} : \mathsf{cmd}(s)$ and $\Sigma; \Gamma; \Delta_2 \vdash S_\mathsf{F} : \mathsf{cmd}(s)$. We can then conclude $\Sigma; \Gamma; \Delta_2 \vdash \mathtt{if}\ e\ \mathtt{then}\ S_\mathsf{T}\ \mathtt{else}\ S_\mathsf{F} : \mathsf{cmd}(s)$ by [T-IF].
- —If [T-SEQ] was used, then $S = S_1; S_2$; from the premise, we know that $\Sigma; \Gamma; \Delta_1 \vdash S_1 : \mathsf{cmd}(s)$ and $\Sigma; \Gamma; \Delta_1 \vdash S_2 : \mathsf{cmd}(s)$. By two applications of the induction hypothesis, we obtain that $\Sigma; \Gamma; \Delta_2 \vdash S_1 : \mathsf{cmd}(s)$ and $\Sigma; \Gamma; \Delta_2 \vdash S_2 : \mathsf{cmd}(s)$. We can then conclude $\Sigma; \Gamma; \Delta_2 \vdash S_1; S_2 : \mathsf{cmd}(s)$ by [T-SEQ].
- —If [T-SSUB] was used, then we know from the premise that $\Sigma; \Gamma; \Delta_1 \vdash S : \mathsf{cmd}(s')$ and $s \sqsubseteq s'$. By the induction hypothesis, we obtain that $\Sigma; \Gamma; \Delta_2 \vdash S : \mathsf{cmd}(s')$. Then, we can conclude $\Sigma; \Gamma; \Delta_2 \vdash S : \mathsf{cmd}(s)$ by [T-SSUB]. □

E Proof of Theorem 3

Proof. By induction on the derivation of $\mathsf{env}_T \vdash \langle S, \mathsf{env}_{SV}\rangle \to_\mathsf{S} \mathsf{env}'_{SV}$. There are four base cases (corresponding to the rules of Figure 6 that have no occurrences of $\to_\mathsf{S}$ in their premise):

- —If [SKIP] was used, then $S = \mathtt{skip}$ and the result is immediate, since skip does not affect env_{SV}.
- —If [WHILE$_\mathsf{F}$] was used, then $S = \mathtt{while}\ e\ \mathtt{do}\ S'$, with e that evaluates to F. The result is immediate, since env_{SV} is not modified in the transition.
- —If [ASSV] was used, then $S = x\ \mathtt{:=}\ e$ and from its premise we know that $\mathsf{env}_{SV} \vdash e \to_\mathsf{e} v$ and $\mathsf{env}'_V = \mathsf{env}_V[x \mapsto v]$. By the premise of [T-ASSV], $\Delta(x) = \mathsf{var}(B_s)$ and $\Sigma; \Gamma; \Delta \vdash e : B_s$.
  - –By Theorem 1, we know that $\Sigma; \Gamma; \Delta \vdash v : B_s$. By Lemma 4 we then conclude that $\Sigma; \Gamma; \Delta \vdash \mathsf{env}_V[x \mapsto v]$.
  - –As $s \not\sqsubseteq s'$, we therefore conclude $\Gamma \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}_S, \mathsf{env}_V[x \mapsto v]$.

—If [ASSF] was used, then $S = \texttt{this}.p \texttt{ := } e$. The argument is the same as above, except that the update affects env_S, rather than env_V.

For the inductive step, we distinguish the last rule used in the inference; we have the following six cases to consider:

—If [SEQ] was used, then $S = S_1 ; S_2$ and from its premise we know that

$$\mathsf{env}_T \vdash \langle S_1, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}''_{SV} \qquad \mathsf{env}_T \vdash \langle S_2, \mathsf{env}''_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}.$$

Moreover, the last rule used to type S is [T-SEQ], whose premise ensures that $\Sigma;\Gamma;\Delta \vdash S_1 : \mathsf{cmd}(s)$ and $\Sigma;\Gamma;\Delta \vdash S_2 : \mathsf{cmd}(s)$. By two applications of the induction hypothesis, the statement holds for both premises; so, we obtain that
– $\Sigma;\Gamma;\Delta \vdash \mathsf{env}''_{SV}$ and $\Sigma;\Gamma;\Delta \vdash \mathsf{env}'_{SV}$,
– $\Gamma;\Delta \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}''_{SV}$ and $\Gamma;\Delta \vdash \mathsf{env}''_{SV} =_{s'} \mathsf{env}'_{SV}$, and we conclude by transitivity.

—If [IF] was used, then $S = \texttt{if } e \texttt{ then } S_{\mathsf{T}} \texttt{ else } S_{\mathsf{F}}$ and from its premise we know that $\mathsf{env}_T \vdash \langle S_b, \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}$, where $b \in \{\mathsf{T}, \mathsf{F}\}$ depending on the evaluation value of e. By [T-IF], we know that $\Sigma;\Gamma;\Delta \vdash S_{\mathsf{T}} : \mathsf{cmd}(s)$ and $\Sigma;\Gamma;\Delta \vdash S_{\mathsf{F}} : \mathsf{cmd}(s)$; regardless of which branch was chosen, the statement then holds by the induction hypothesis.

—If [WHILE$_{\mathsf{T}}$] was used, then $S = \texttt{while } e \texttt{ do } S'$, with e that evaluates to T. From its premise, we know that $\mathsf{env}_T \vdash \langle S', \mathsf{env}_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}''_{SV}$ and $\mathsf{env}_T \vdash \langle \texttt{while } e \texttt{ do } S', \mathsf{env}''_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}$. By the premise of rule [T-WHILE], we know that $\Sigma;\Gamma;\Delta \vdash S' : \mathsf{cmd}(s)$. The statement then holds by two applications of the induction hypothesis.

—If [DECV] was used, then $S = \mathsf{var}(B_{s''}) \texttt{ } x \texttt{ := } e \texttt{ in } S'$, where $x \notin \mathsf{dom}(\mathsf{env}'_V)$. From its premise we know that

$$\mathsf{env}_{SV} \vdash e \to_{\mathsf{e}} v \qquad \mathsf{env}_T \vdash \langle S', \mathsf{env}_S; \{(x,v)\} \cup \mathsf{env}_V\rangle \to_{\mathsf{S}} \mathsf{env}'_S; \{(x,v')\} \cup \mathsf{env}'_V$$

By the premises of [T-DECV], we have that $\Sigma;\Gamma;\Delta \vdash e : B_{s''}$ and $\Sigma;\Gamma;\Delta, x : \mathsf{var}(B_{s''}) \vdash S' : \mathsf{cmd}(s)$. By Theorem 1, we know that $\Sigma;\Gamma;\Delta \vdash v : B_{s''}$. By Lemma 3, we have that $\Sigma;\Gamma;\Delta, x : \mathsf{var}(B_{s''}) \vdash \mathsf{env}_S; \{(x,v)\} \cup \mathsf{env}_V$. We can then apply the induction hypothesis to conclude that
– $\Sigma;\Gamma;\Delta, x : \mathsf{var}(B_{s''}) \vdash \mathsf{env}'_S; \{(x,v')\} \cup \mathsf{env}'_V$.
– $\Gamma;\Delta, x : \mathsf{var}(B_{s''}) \vdash \mathsf{env}_S; \{(x,v)\} \cup \mathsf{env}_V =_{s'} \mathsf{env}'_S; \{(x,v')\} \cup \mathsf{env}'_V$.
Finally, by Lemma 2, we can then conclude $\Sigma;\Gamma;\Delta \vdash \mathsf{env}'_{SV}$ and $\Gamma;\Delta \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}'_{SV}$, as required.

—If [CALL] was used, then $S = e_1.f(\tilde{e})\$e_2$ and it was typed by using [T-CALL]. From the premise of [CALL], we know that $\mathsf{env}_V(\texttt{this}) = X$ (the caller), $\mathsf{env}_{SV} \vdash e_1 \to_{\mathsf{e}} Y$ (the callee), and there are two writes to the `balance` fields of caller and callee (which are typed as $\mathsf{var}(\mathsf{int}, s_2)$ and $\mathsf{var}(\mathsf{int}, s_3)$, for $s \sqsubseteq s_2, s_3$), that generate the new environment env''_S (that only differs from env_S for the values assigned to the balances); so, $\Gamma \vdash \mathsf{env}_S =_{s'} \mathsf{env}''_S$. Moreover, $\mathsf{env}_{SV} \vdash e_2 \to_{\mathsf{e}} z$, $\mathsf{env}_{SV} \vdash \tilde{e} \to_{\mathsf{e}} \tilde{v}$ and $\mathsf{env}_T \vdash \langle S', \mathsf{env}''_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}'_{SV}$, where S' is the body of the method and $\mathsf{env}''_V = \{(\texttt{this}, Y), (\texttt{sender}, X), (\texttt{value}, z), (\tilde{x}, \tilde{v})\}$.

By the premises of [T-CALL], we know that

$$\Sigma;\Gamma;\Delta \vdash e_1 : I^Y_s \qquad \Gamma(I^Y)(f) = \mathsf{proc}(\widetilde{B_s}) : s \qquad \Sigma;\Gamma;\Delta \vdash \tilde{e} : \widetilde{B_s}$$
$$\Sigma;\Gamma;\Delta \vdash e_2 : \mathsf{int}_s \qquad \Delta(\texttt{this}) = \mathsf{var}(I^X, s_1)$$

Moreover, from [T-DEC-M] we also know that $\Sigma;\Gamma;\Delta' \vdash S' : \mathsf{cmd}(s)$, where $\Delta' = \texttt{this} : \mathsf{var}(I^Y_s), \texttt{sender} : \mathsf{var}(I^\top, s_\top), \texttt{value} : \mathsf{var}(\mathsf{int}_s), \tilde{x} : \widetilde{\mathsf{var}(B_s)}$, where $\Sigma \vdash (I^X, s_1) <: (I^\top, s_\top)$. Hence, $\Sigma;\Gamma;\Delta' \vdash \mathsf{env}''_V$ holds by [T-ENV-V]. By the induction hypothesis, we can then conclude that

– $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_{SV}$, and therefore in particular also $\Sigma; \Gamma; \emptyset \vdash \mathsf{env}'_S$, which trivially implies $\Sigma; \Gamma; \Delta \vdash \mathsf{env}'_S, \mathsf{env}_V$.
– $\Gamma; \Delta' \vdash \mathsf{env}''_{SV} =_{s'} \mathsf{env}'_{SV}$, and therefore in particular also $\Gamma \vdash \mathsf{env}_S =_{s'} \mathsf{env}'_S$, which trivially implies $\Gamma; \Delta \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}'_S, \mathsf{env}_V$.

— If [DCALL] was used, then $S = \texttt{dcall}\ e.f(\tilde{e})$; from its premise, we know that

$$\mathsf{env}_{SV} \vdash e \to_{\mathrm{e}} Y \qquad \mathsf{env}_{SV} \vdash \tilde{e} \to_{\mathrm{e}} \tilde{v} \qquad \mathsf{env}_T \vdash \langle S', \mathsf{env}_S, \mathsf{env}''_V \rangle \to_{\mathrm{S}} \mathsf{env}'_{SV}$$

where S' is the body of the method and

$$\mathsf{env}''_V = \{(\texttt{this}, \mathsf{env}_V(\texttt{this})), (\texttt{sender}, \mathsf{env}_V(\texttt{sender})), (\texttt{value}, \mathsf{env}_V(\texttt{value})), (\tilde{x}, \tilde{v})\}.$$

By the premises of [T-DCALL], we know that

$$\Sigma; \Gamma; \Delta \vdash e : I^Y_s \qquad \Gamma(I^Y)(f) = \mathsf{proc}(\widetilde{B_s}) : s \qquad \Sigma; \Gamma; \Delta \vdash \tilde{e} : \widetilde{B_s}$$
$$\Delta(\texttt{this}) = \mathsf{var}(I^X, s_1) \qquad \Sigma \vdash (I^X, s_1) <: (I^Y, s)$$
$$\forall q \in (\mathrm{dom}\,(\Gamma(I^Y)) \cap (\mathsf{FNames} \cup \{\texttt{balance}\})).\ \Gamma(I^Y)(q) = \Gamma(I^X)(q)$$

Moreover, from [T-DEC-M] we also know that $\Sigma; \Gamma; \Delta'' \vdash S' : \mathsf{cmd}(s)$, where

$$\Delta'' = \texttt{this} : \mathsf{var}(I^Y_s), \texttt{sender} : \mathsf{var}(I^\top, s_\top), \texttt{value} : \mathsf{var}(\mathsf{int}_s), \tilde{x} : \widetilde{\mathsf{var}(B_s)}.$$

Now, let

$$\Delta' = \texttt{this} : \mathsf{var}(I^X, s_1), \texttt{sender} : \mathsf{var}(I^\top, s_\top), \texttt{value} : \mathsf{var}(\mathsf{int}_s), \tilde{x} : \widetilde{\mathsf{var}(B_s)}.$$

By Lemma 6 we can then conclude that $\Sigma; \Gamma; \Delta' \vdash S' : \mathsf{cmd}(s)$.

Since we know that $\Sigma \vdash (I^X, s_1) <: (I^Y, s)$, we can also conclude $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}''_V$ by [T-ENV-V]. Now, by the induction hypothesis, we can conclude that

– $\Sigma; \Gamma; \Delta' \vdash \mathsf{env}'_{SV}$, and therefore in particular also $\Sigma; \Gamma; \emptyset \vdash \mathsf{env}'_S$, which trivially implies $\Sigma; \Gamma; \Delta \vdash \mathsf{env}'_S, \mathsf{env}_V$.
– $\Gamma; \Delta' \vdash \mathsf{env}_S, \mathsf{env}''_V =_{s'} \mathsf{env}'_{SV}$, and therefore in particular $\Gamma \vdash \mathsf{env}_S =_{s'} \mathsf{env}'_S$, which trivially implies $\Gamma; \Delta \vdash \mathsf{env}_{SV} =_{s'} \mathsf{env}'_S, \mathsf{env}_V$. □

F Details of the Proof of Theorem 4

Proof. We consider here the case when $s_1 \sqsubseteq s_2$ and we proceed by induction on the sum of the lengths of the inferences for $\mathsf{env}_T \vdash \langle S, \mathsf{env}^1_{SV} \rangle \to_{\mathrm{S}} \mathsf{env}^{1'}_{SV}$ and $\mathsf{env}_T \vdash \langle S, \mathsf{env}^2_{SV} \rangle \to_{\mathrm{S}} \mathsf{env}^{2'}_{SV}$; as we proved in the proof sketch of this Theorem, both inferences terminate with the very same rule of Figure 6. There are four base cases (corresponding to the rules of Figure 6 that have no occurrence of $\to_{\mathrm{S}}$ in their premise):

— If [SKIP] was used to infer both transitions, then $S = \texttt{skip}$ and the result is immediate, since skip does not affect env^1_{SV} and env^2_{SV}.
— If [WHILE$_{\mathrm{F}}$] is used to conclude both transitions, then $S = \texttt{while}\ e\ \texttt{do}\ S'$ and the result is immediate, since $\mathsf{env}^{1'}_{SV} = \mathsf{env}^1_{SV}$ and $\mathsf{env}^{2'}_{SV} = \mathsf{env}^2_{SV}$.
— If [ASSV] was used to infer both transitions, then $S = x\ \texttt{:=}\ e$. From the premises of that rule, we know that $\mathsf{env}^1_{SV} \vdash e \to_{\mathrm{e}} v_1$ and $\mathsf{env}^2_{SV} \vdash e \to_{\mathrm{e}} v_2$; thus, $\mathsf{env}^{1'}_{SV} = \mathsf{env}^1_S, \mathsf{env}^1_V[x \mapsto v_1]$ and $\mathsf{env}^{2'}_{SV} = \mathsf{env}^2_S, \mathsf{env}^2_V[x \mapsto v_2]$. By the premise of [T-ASSV], $\Delta(x) = \mathsf{var}(B, s_1)$ and $\Sigma; \Gamma; \Delta \vdash e : (B, s_1)$. Since by assumption $s_1 \sqsubseteq s_2$ and $\Gamma; \Delta \vdash \mathsf{env}^1_{SV} =_{s_2} \mathsf{env}^2_{SV}$, by Theorem 2 we can conclude that $v_1 = v_2 = v$. Thus, $\Gamma; \Delta \vdash \mathsf{env}^1_S, \mathsf{env}^1_V[x \mapsto v] =_{s_2} \mathsf{env}^2_S, \mathsf{env}^2_V[x \mapsto v]$ also holds.
— If [ASSF] was used, then $S = \texttt{this}.p\ \texttt{:=}\ e$. The argument is then the same as above, except that it is env_S, rather than env_V, that is updated.

For the inductive step, we distinguish the last rule used in both inferences (as we argued above, the two transitions have been inferred by using the same last rule). We have the following six cases to consider:

—If [SEQ] was used to conclude both transitions, then $S = S_1 ; S_2$. From the premises of that rule, we know that

$$\mathsf{env}_T \vdash \langle S_1, \mathsf{env}^1_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{1''}_{SV} \qquad \mathsf{env}_T \vdash \left\langle S_2, \mathsf{env}^{1''}_{SV}\right\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV}$$

and

$$\mathsf{env}_T \vdash \langle S_1, \mathsf{env}^2_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{2''}_{SV} \qquad \mathsf{env}_T \vdash \left\langle S_2, \mathsf{env}^{2''}_{SV}\right\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV}$$

Moreover, from the premises of [T-SEQ], we know that $\Sigma; \Gamma; \Delta \vdash S_1 : \mathsf{cmd}(s_1)$ and $\Sigma; \Gamma; \Delta \vdash S_2 : \mathsf{cmd}(s_1)$. By two applications of the induction hypothesis, we conclude that $\Gamma; \Delta \vdash \mathsf{env}^{1''}_{SV} =_{s_2} \mathsf{env}^{2''}_{SV}$ and $\Gamma; \Delta \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$, as required.

—If [IF] was the last rule used in both inferences, then $S = \texttt{if}\ e\ \texttt{then}\ S_{\mathsf{T}}\ \texttt{else}\ S_{\mathsf{F}}$. Like in the case of the `while` discussed at the beginning of this proof, we have that $b_1 = b_2 = b \in \{\mathsf{T}, \mathsf{F}\}$, where $\mathsf{env}^1_{SV} \vdash e \to_{\mathsf{e}} b_1$ and $\mathsf{env}^2_{SV} \vdash e \to_{\mathsf{e}} b_2$; thus, the same branch S_b is chosen in both inferences, that is $\mathsf{env}_T \vdash \langle S_b, \mathsf{env}^1_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV}$ and $\mathsf{env}_T \vdash \langle S_b, \mathsf{env}^2_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV}$. Regardless of the value of b, from the premise of [T-IF] we know that $\Sigma; \Gamma; \Delta \vdash e : s_1$ and $\Gamma \vdash S_b : \mathsf{cmd}(s_1)$. The statement then holds by the induction hypothesis.

—If [WHILE_{T}] is used to conclude both the inferences, then $S = \texttt{while}\ e\ \texttt{do}\ S'$ and from the premises of that rule we know that

$$\mathsf{env}_T \vdash \langle S', \mathsf{env}^1_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{1''}_{SV} \qquad \mathsf{env}_T \vdash \left\langle \texttt{while}\ e\ \texttt{do}\ S', \mathsf{env}^{1''}_{SV}\right\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV}$$

and

$$\mathsf{env}_T \vdash \langle S', \mathsf{env}^2_{SV}\rangle \to_{\mathsf{S}} \mathsf{env}^{2''}_{SV} \qquad \mathsf{env}_T \vdash \left\langle \texttt{while}\ e\ \texttt{do}\ S', \mathsf{env}^{2''}_{SV}\right\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV}$$

Moreover, by the premise of [T-WHILE], we know that $\Sigma; \Gamma; \Delta \vdash e : s_1$ and $\Sigma; \Gamma; \Delta \vdash S' : \mathsf{cmd}(s_1)$. The statement then holds by two applications of the induction hypothesis, like in the case for [SEQ] above.

—If [DECV] was the last rule in both inferences, then $S = \mathsf{var}(B)\ x\ \texttt{:=}\ e\ \texttt{in}\ S'$; from the premises of that rule, we have that

$$\mathsf{env}^1_{SV} \vdash e \to_{\mathsf{e}} v_1 \qquad \mathsf{env}_T \vdash \langle S', \mathsf{env}^1_S, \{(x, v_1)\} \cup \mathsf{env}^1_V\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_S, \{(x, v'_1)\} \cup \mathsf{env}^{1'}_V$$

and

$$\mathsf{env}^2_{SV} \vdash e \to_{\mathsf{e}} v_2 \qquad \mathsf{env}_T \vdash \langle S', \mathsf{env}^2_S, \{(x, v_2)\} \cup \mathsf{env}^2_V\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_S, \{(x, v'_2)\} \cup \mathsf{env}^{2'}_V$$

where $x \notin \mathrm{dom}\,(\mathsf{env}^1_V) \cup \mathrm{dom}\,(\mathsf{env}^2_V)$.
From the premise of [T-DECV], we know that $\Sigma; \Gamma; \Delta \vdash e : B_s$ and $\Sigma; \Gamma; \Delta, x : \mathsf{var}(B_s) \vdash S' : \mathsf{cmd}(s_1)$. In order to apply the induction hypothesis, we must show that

$$\Gamma; \Delta, x : \mathsf{var}(B_s) \vdash \mathsf{env}^1_S, \{(x, v_1)\} \cup \mathsf{env}^1_V =_{s_2} \mathsf{env}^2_S, \{(x, v_2)\} \cup \mathsf{env}^2_V \tag{6}$$

If $s \sqsubseteq s_2$, then (6) holds by Theorem 2: indeed, $v_1 = v_2 = v$, since all variables read within e are of a lower level than s_2, and by assumption, env^1_{SV} and env^2_{SV} agree on all values up to, and including, s_2. If $s \not\sqsubseteq s_2$, then (6) holds by definition of the $\Gamma; \Delta \vdash \cdot =_s \cdot$ relation.
Hence, by the induction hypothesis we have that $\Gamma; \Delta, x : \mathsf{var}(B_s) \vdash \mathsf{env}^{1'}_S, \{(x, v'_1)\} \cup \mathsf{env}^{1'}_V =_{s_2} \mathsf{env}^{2'}_S, \{(x, v'_2)\} \cup \mathsf{env}^{2'}_V$; by Lemma 2, we can then conclude $\Gamma; \Delta \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$, as required.

—If [CALL] was the last rule, then $S = e_1.f(\tilde{e})\$e_2$ and so it has been typed by using [T-CALL]. First, from the premises of [CALL] we know that $\mathsf{env}^1_V(\mathtt{this}) = X_1$ and $\mathsf{env}^2_V(\mathtt{this}) = X_2$. From the premise of [T-CALL], we have that $\Delta(\mathtt{this}) = \mathsf{var}(I^X, s'_1)$ with $s'_1 \sqsubseteq s_1$. Since $\Delta \vdash \mathsf{env}^1_V =_{s_2} \mathsf{env}^2_V$ and $s_1 \sqsubseteq s_2$, we can conclude that $X_1 = X_2 = X$, by [EQ-ENV$_V$].
Second, from the premises of [CALL] and [T-CALL], we have that

$$\mathsf{env}^1_{SV} \vdash e_1 \to_{\mathsf{e}} Y_1 \qquad \mathsf{env}^2_{SV} \vdash e_1 \to_{\mathsf{e}} Y_2 \qquad \Sigma;\Gamma;\Delta \vdash e_1 : (I^Y, s_1)$$
$$\mathsf{env}^1_{SV} \vdash e_2 \to_{\mathsf{e}} z_1 \qquad \mathsf{env}^2_{SV} \vdash e_2 \to_{\mathsf{e}} z_2 \qquad \Sigma;\Gamma;\Delta \vdash e_2 : (\mathsf{int}, s_1)$$

By Theorem 2, we conclude that $Y_1 = Y_2 = Y$ and $z_1 = z_2 = z$, since we are assuming that $s_1 \sqsubseteq s_2$.
Third, for the list of actual parameters, fix an arbitrary expression $e_i \in \tilde{e}$. By the premises of [CALL] and [T-CALL], we have that

$$\mathsf{env}^1_{SV} \vdash e_i \to_{\mathsf{e}} v^1_i \qquad \mathsf{env}^2_{SV} \vdash e_i \to_{\mathsf{e}} v^2_i \qquad \Sigma;\Gamma;\Delta \vdash e_i : (B_i, s_i)$$

If $s_i \sqsubseteq s_2$, then by Theorem 2 we have that $v^1_i = v^2_i$. Otherwise, it may be the case that $v^1_i \neq v^2_i$, but since $s_i \not\sqsubseteq s_2$, then this assignment will still satisfy the condition that the variable-environments will agree up to level s_2.
Being the callee and the caller the same in both transitions, by [EQ-ENV$_F$] we have that $\Gamma \vdash_{I^X} \mathsf{env}^{1X}_F =_{s_2} \mathsf{env}^{2X}_F$ and $\Gamma \vdash_{I^Y} \mathsf{env}^{1Y}_F =_{s_2} \mathsf{env}^{2Y}_F$, where $\mathsf{env}^{iZ}_F = \mathsf{env}^i_S(Z)$ for $i \in \{1,2\}$ and $Z \in \{X, Y\}$. Since [T-CALL] entails that $\Gamma(I^X)(\mathtt{balance}) = \mathsf{var}(\mathsf{int}, s'_2)$ and $\Gamma(I^Y)(\mathtt{balance}) = \mathsf{var}(\mathsf{int}, s'_3)$, with $s_1 \sqsubseteq s'_2, s'_3$, we have that

$$\Gamma \vdash \mathsf{env}^{1''}_S =_{s_2} \mathsf{env}^{2''}_S \tag{7}$$

where

$$\mathsf{env}^{1''}_S = \mathsf{env}^1_S[X \mapsto \mathsf{env}^{1X}_F[\mathtt{balance\ -=}\ z]][Y \mapsto \mathsf{env}^{1Y}_F[\mathtt{balance\ +=}\ z]]$$
$$\mathsf{env}^{2''}_S = \mathsf{env}^2_S[X \mapsto \mathsf{env}^{2X}_F[\mathtt{balance\ -=}\ z]][Y \mapsto \mathsf{env}^{2Y}_F[\mathtt{balance\ +=}\ z]]$$

From the premises of [CALL] we have that

$$\mathsf{env}_T \vdash \left\langle S', \mathsf{env}^{1''}_{SV} \right\rangle \to_{\mathsf{S}} \mathsf{env}^{1'}_{SV} \qquad \mathsf{env}_T \vdash \left\langle S', \mathsf{env}^{2''}_{SV} \right\rangle \to_{\mathsf{S}} \mathsf{env}^{2'}_{SV} \tag{8}$$

where $\mathsf{env}_T(Y)(f) = (\tilde{x}, S')$ and

$$\mathsf{env}^{1''}_V = \{(\mathtt{this}, Y), (\mathtt{sender}, X), (\mathtt{value}, z), (x_1, v^1_1), \ldots, (x_h, v^1_h)\}$$
$$\mathsf{env}^{2''}_V = \{(\mathtt{this}, Y), (\mathtt{sender}, X), (\mathtt{value}, z), (x_1, v^2_1), \ldots, (x_h, v^2_h)\}$$

From what we proved above, by [EQ-ENV$_V$] it easily follows that

$$\Delta' \vdash \mathsf{env}^{1''}_V =_{s_2} \mathsf{env}^{2''}_V \tag{9}$$

where $\Delta' = \mathtt{this} : \mathsf{var}(I^Y, s_1), \mathtt{sender} : \mathsf{var}(I^\top, s_\top), \mathtt{value} : \mathsf{var}(\mathsf{int}, s_1), \tilde{x} : \widetilde{\mathsf{var}(B_s)}$ and $\Sigma \vdash (I^X, s'_1) <: (I^\top, s_\top)$.
Finally, from the premise of [T-DEC-M], we have that $\Sigma;\Gamma;\Delta' \vdash S' : \mathsf{cmd}(s_1)$. This, together with $\Gamma \vdash \mathsf{env}^1_T =_{s_2} \mathsf{env}^2_T$ (that holds by hypothesis), $\Gamma;\Delta' \vdash \mathsf{env}^{1''}_{SV} =_{s_2} \mathsf{env}^{2''}_{SV}$ (that holds because of (7) and (9)), and (8), allows us to apply the inductive hypothesis and obtain that $\Gamma;\Delta' \vdash \mathsf{env}^{1'}_{SV} =_{s_2} \mathsf{env}^{2'}_{SV}$. This allows us to conclude that $\Gamma;\Delta \vdash \mathsf{env}^{1'}_S, \mathsf{env}^1_V =_{s_2} \mathsf{env}^{2'}_S, \mathsf{env}^2_V$, as required.

—If [DCALL] was used, then $S = \texttt{dcall}\ \ e.f(\tilde{e})$ and so it has been typed by using [T-DCALL]. By the premises of [T-DCALL], we know that

$$\Sigma;\Gamma;\Delta \vdash e : (I^Y, s_1) \qquad \Gamma(I^Y)(f) = \mathsf{proc}(\widetilde{B_s}) : s_1 \qquad \Sigma;\Gamma;\Delta \vdash \tilde{e} : \widetilde{B_s}$$
$$\Delta(\texttt{this}) = \mathsf{var}(I^X, s_1') \qquad \Sigma \vdash : (I^X, s_1') <: (I^Y, s_1)$$

From $\Delta \vdash \mathsf{env}_V^1 =_{s_2} \mathsf{env}_V^2$, we obtain that $\mathsf{env}_V^1(\texttt{this}) = \mathsf{env}_V^2(\texttt{this})$, being $s_1' \sqsubseteq s_1$. Moreover, like in the case for [CALL], we can prove that the callee (given by evaluating e and yielding Y) is the same in the two transitions; the same happens for the actual parameters (obtained by evaluating $\tilde{e}$ and yielding $\widetilde{v_1}$ and $\widetilde{v_2}$) whose level is smaller than or equal to s_2. So, by [T-ENV-V] and [EQ-ENV$_V$]

$$\Sigma;\Gamma;\Delta' \vdash \mathsf{env}_V^{1''} \qquad \Sigma;\Gamma;\Delta' \vdash \mathsf{env}_V^{2''} \qquad \Delta' \vdash \mathsf{env}_V^{1''} =_{s_2} \mathsf{env}_V^{2''} \tag{10}$$

where

$$\Delta' = \texttt{this} : \mathsf{var}(I^Y, s_1), \texttt{sender} : \mathsf{var}(I^\top, s_\top), \texttt{value} : \mathsf{var}(\mathsf{int}, s_1), \tilde{x} : \widetilde{\mathsf{var}(B_s)}$$
$$\mathsf{env}_V^{1''} = \{(\texttt{this}, \mathsf{env}_V^1(\texttt{this})), (\texttt{sender}, \mathsf{env}_V^1(\texttt{sender})), (\texttt{value}, \mathsf{env}_V^1(\texttt{value})), (\tilde{x}, \widetilde{v_1})\}$$
$$\mathsf{env}_V^{2''} = \{(\texttt{this}, \mathsf{env}_V^2(\texttt{this})), (\texttt{sender}, \mathsf{env}_V^2(\texttt{sender})), (\texttt{value}, \mathsf{env}_V^2(\texttt{value})), (\tilde{x}, \widetilde{v_2})\}$$

and $\mathsf{env}_T(Y)(f) = (\tilde{x}, S')$. From the premises of [DCALL] we have that

$$\mathsf{env}_T \vdash \left\langle S', \mathsf{env}_S^1, \mathsf{env}_V^{1''} \right\rangle \to_{\mathrm{S}} \mathsf{env}_{SV}^{1'} \qquad \mathsf{env}_T \vdash \left\langle S', \mathsf{env}_S^2, \mathsf{env}_V^{2''} \right\rangle \to_{\mathrm{S}} \mathsf{env}_{SV}^{2'}$$

Moreover, from [T-DEC-M] we also know that $\Sigma;\Gamma;\Delta' \vdash S' : \mathsf{cmd}(s_1)$; so we can apply the induction hypothesis (thanks to (10)) to obtain $\Gamma;\Delta' \vdash \mathsf{env}_S, \mathsf{env}_V'' =_{s_2} \mathsf{env}_{SV}'$ that trivially implies $\Gamma;\Delta \vdash \mathsf{env}_{SV} =_{s_2} \mathsf{env}_S', \mathsf{env}_V$, as desired. □

G Summary of Symbols and Notations

Name	Symbol	Set	Defined in
Variable names	x, y	VNames	Figure 2
Magic variables (`this`, `sender`, `value`)	m	MVar	Figure 2
All variables	k		Section 2.2.2
Field names	p	FNames	Figure 2
All fields (including `balance`)	q		Section 2.2.2
Method names	f, g	MNames	Figure 2
Address names	A, X, Y, Z	ANames	Figure 2
Names (all the previous ones)	n	$\mathcal{N}$	Definition 7
Interface names	I	INames	Definition 7
Integers	z	$\mathbb{Z}$	Section 2.1
Booleans	b	$\mathbb{B}$	Section 2.1
Values	v	Val	Figure 2
Field declarations	DF	Dec_F	Figure 2
Method declarations	DM	Dec_M	Figure 2
Contract declarations	DC	Dec_C	Figure 2
Assignable containers	L	LVal	Figure 2
Expressions	e	Exp	Figure 2
Statements	S	Stm	Figure 2
Blockchains	BC	$\mathcal{BC}$	Definition 1
Transactions	τ	Tr	Definition 1
Traces	π		Definition 3
Variables environments	env_V	Env_V	Definition 2
Fields environments	env_F	Env_F	Definition 2
Methods environments	env_M	Env_M	Definition 2
States	env_S	Env_S	Definition 2
Method tables	env_T	Env_T	Definition 2
Operation evaluation	$\rightarrow_{\text{op}}$		Assumed
Semantics of declarations	$\rightarrow_{\text{DF}}, \rightarrow_{\text{DM}}, \rightarrow_{\text{DC}}$		Figure 4
Semantics of expressions	$\rightarrow_{\text{e}}$		Figure 5
Semantics of statements	$\rightarrow_{\text{S}}$		Figure 6
Semantics of blockchains	$\rightarrow_{\text{B}}$		Figure 7
Trace semantics	$\overset{\pi}{\rightarrow}_{\text{S}}, \overset{\pi}{\rightarrow}_{\text{B}}$		Definition 3

Name	Symbol		Defined in
Security levels (partially ordered with $\sqsubseteq$)	s	$\mathcal{S}$	Section 4.1
Base types	B	$\mathcal{B}$	Definition 7
Types	T	$\mathcal{T}$	Definition 7
Type environments	Γ, Δ	$\mathcal{E}$	Definition 7
Interfaces hierarchies	Σ		Definition 7
Consistency of Σ	$\Sigma; \Gamma \vdash \Sigma'$		Figure 8
Subtyping	$\Sigma \vdash T_1 <: T_2$		Figures 9 and 10
Typing of expressions	$\Sigma; \Gamma; \Delta \vdash e : B_s$		Figure 11
Typing of statements	$\Sigma; \Gamma; \Delta \vdash S : \mathsf{cmd}(s)$		Figures 12 and 13
Typing of environments	$\Sigma; \Gamma; \Delta \vdash_{(X)} \mathsf{env}$		Figure 15
Typing of declarations	$\Sigma; \Gamma; \Delta \vdash_{(X)} D$		Figure 16
Typing of transactions	$\Sigma; \Gamma; \Delta \vdash_{(X)} \tau : \mathsf{cmd}(s)$		Section 4.5
s-equivalence	$=_s$		Figure 17

Received 5 March 2025; revised 7 August 2025; accepted 16 December 2025