Formal Verification of Atomicity Requirements for Smart Contracts

Preprint · September 2020

CITATIONS
0

READS
334

5 authors, including:

Ximeng Li
Capital Normal University
32 PUBLICATIONS **49** CITATIONS
SEE PROFILE

Zhiping Shi
Capital Normal University
149 PUBLICATIONS **1,010** CITATIONS
SEE PROFILE

Formal Verification of Atomicity Requirements for Smart Contracts

Ning Han[1], Ximeng Li[1,3] Guohui Wang[2], Zhiping Shi[1], Yong Guan[3]

[1] Beijing Key Laboratory of Electronic System Reliability and Prognostics
15238483068@163.com, shizp@cnu.edu.cn
[2] Beijing Engineering Research Center of High Reliable Embedded System
ghwang@cnu.edu.cn
[3] Beijing Advanced Innovation Center for Imaging Theory and Technology
lixm@cnu.edu.cn, guanyong@cnu.edu.cn

Capital Normal University, Beijing, China

Abstract. Smart contracts are notoriously vulnerable to bugs and loopholes. This is due largely to an unusual combination of features: reentrant calls, transfer-triggered code execution, the way exceptions are propagated, etc. Numerous validation techniques have been developed to ensure the safety and security of smart contracts. An important class of problems dealt with is related to the atomic performance of actions such as contract calls and state updates. In this paper, we examine the major existing atomicity-related criteria for the safety and security of smart contracts. We then propose an atomicity criterion and argue about its advantages. Furthermore, we develop a Hoare-style program logic that is capable of verifying the fulfillment of safety requirements by smart contracts, including the satisfaction of the proposed criterion. The program logic is developed and proven sound for a core Solidity-like language, which supports reentrant calls, ether transfers, and exception handling.

1 Introduction

Blockchains are distributed digital ledgers containing records of user data and activities [33]. The blockchain technology provides multiple desired features such as distributed consensus, decentralized management and control, the difficulty in corrupting data and fake data, the traceability of provenance, etc. This technology has the potential to play a key role in the effective and efficient management of trust relationship in the information era.

Since the advent of Ethereum [32], programmability has become an indispensable feature of blockchain systems. Programs implementing smart contracts [30] can be written to define the logic of transactions over a blockchain system. Programmability greatly eases the development of a wide spectrum of blockchain applications, ranging from digital currency to online gaming.

There has been a high level of activity in the development of smart contracts. However, a great number of bugs and loopholes have been found to exist

This is preprint of an article accepted for presentation at the 18th Asian Symposium on Programming Languages and Systems (APLAS'2020).

in the smart contracts deployed. These bugs and loopholes put on-chain digital assets at stake [5, 1]. The safety and security problems of smart contracts have led to extensive recent efforts related to the validation of smart contracts (as recently surveyed in [31]). The bulk of the existing work is focused on bug detection and verification [24, 9, 22, 25, 20, 34, 15, 3, 7, 12, 17, 11, 26, 27], semantic foundation [19, 18, 16, 13, 6, 21], or language design [28, 8, 10]. Relatively little attention has been directed to the discussion about the safety and security requirements themselves. The mis-specification of requirements and criteria could lead to undetected flaws or false alarms, even if the analysis is sound and precise with respect to the specified requirements.

An important class of requirements for the safety and security of smart contracts is the atomicity of operations – a group of related operations should be performed in an all-or-nothing fashion. A typical problem is as follows. A contract may fail to transfer an amount of ether, yet deduct this amount from its bookkeeping records, leading to locked funds. Atomicity problems also arise in other scenarios than related to digital currency. Consider a contract that may update the sales figures of a product, without updating its stock. Atomicity problems significantly harm the integrity of the business logics of smart contracts.

In this paper, we propose an atomicity criterion (Sect. 2) that can be used to specify the updates to state variables and the performance of function calls that must happen in sync, in smart contract transactions. The criterion is flexible to use because it supports the specification of which ones of all the potential variable updates and function calls in a smart contract are of concern. Hence, the criterion can be tailored to capture application-specific atomicity requirements.

Currently, the bulk of the existing techniques for the source-level verification of smart contracts does not come with an emphasis on soundness with respect to a concrete language semantics. In this work, we devise a Hoare-style program logic (Sect. 4) for the verification of smart contracts, covering the atomicity criterion we propose, as well as other safety criteria. We establish the soundness of the program logic with respect to a Solidity-like language (Sect. 3), for which we carefully formulate the typing disciplines and a formal semantics.

The main contributions of this paper are:

1. an atomicity criterion for smart contracts supporting the specification of the state updates, as well as function calls, that must happen in atomic batches,
2. a Hoare-style program logic supporting the source-level verification of safety requirements on smart contracts, especially the proposed atomicity criterion.

The proposed program logic provides the vocabulary for referring to, and reasoning about the initial and current values of state variables, the builtin balances of accounts, the caller address, the amount of ether transfered with calls, the presence of successful calls within and between contracts, and the presence of uncaught exceptions. It employs forward-style inference rules for assignments to variables and mapping elements, which enables a more natural style of reasoning and provides a basis for automation. It leverages contract invariants to capture the effect of calls with reentrancy possibilities.

Both the proposed atomicity criterion and program logic are illustrated (Sect. 5) using examples of practical relevance, including a batch-transfer function for the refund of smart contract users.

2 Atomicity Criteria for Smart Contracts

We introduce the proposed atomicity criterion with the help of a simple example. We then discuss major related atomicity criteria in the literature. We use the smart contract language defined in the present work for this discussion, to avoid potential confusion arising from the switch between different languages.

```
fun withdraw(;b:U256) ret x:U256
{
  b := bal;
  if (b = 0) { throw };
  bal := 0;
  try { caller.transfer(b) }
  catch { bal := b; err := err + 1 }
}
```

Fig. 1. The function `withdraw`

The smart contract function `withdraw` in Fig. 1 allows a fixed amount of ether to be withdrawn once. In this function, *bal* is a state variable belonging to the overall contract. It records the amount of ether that can be withdrawn (i.e., the balance). This amount is retrieved in the local variable b, which is declared after the `;` in the first line. If a previous withdrawal has already succeeded (b `= 0` in the conditional), then the call ends with an exception. Otherwise, the bookkeeping balance *bal* is zeroed, and the amount of ether to be withdrawn is transfered to the caller. Exceptions occurring in the transfer are caught and handled by resetting *bal* to its initial value, and incrementing the state variable *err*. This state variable is used to log the number of errors occurring with the ether withdrawal attempts by calling the function `withdraw`.

A key point about the safety of the function `withdraw` is that the actual transfer of ether is kept in sync with the update of the bookkeeping balance *bal*. More concretely, when the execution of `withdraw` is completed, the transfer has been successfully performed, if and only if *bal* has been updated to zero.

There are two further aspects to be noted about `withdraw` in Fig. 1. Firstly, when stating the atomicity requirement, it is more convenient to be concerned with the overall value change of the variable *bal* (e.g., to zero) than with the execution of a specific assignment to *bal* (e.g., *bal*:=b). That is, it is more convenient to consider semantic updates (i.e., overall value changes) of state variables, rather than syntactic updates (e.g., particular assignments to state variables) of them. Secondly, although *err* is also a state variable of the contract like *bal* is, the update of *err* is not supposed to happen in sync with the update of *bal*. Hence, the atomicity of a transaction does not imply that the update of all the state variables should happen in one atomic batch.

In this work, we deal with the formal verification of the following atomicity criterion for smart contracts.

> *For a number of semantic updates to specific state variables, as well as calls to specific functions, either all of them happen or none of them happen, in the execution of a smart contract function.*

For the execution of `withdraw` in Fig. 1, it can be shown that either *bal* is updated, the builtin balance [32] of the contract is updated, and the call to transfer is also successfully performed, or none of the above happens.

Related Atomicity Criteria In the literature, there are two major criteria used to detect atomicity problems in smart contracts that are related to ours.

The first criterion is to check whether write accesses to state variables happen only after successful ether transfers (e.g., [22]). The rationale is that if a transfer fails, the builtin balance of the contract does not change, and, hence, no bookkeeping by means of updating a state variable should be attempted. This criterion is simple to understand, and can be efficiently implemented. On the other hand, this criterion is not met by the function `withdraw`. This is because a write to the state variable *bal* (*bal* := *b*) happens even if the transfer fails. However, it is inappropriate to report this write as causing an atomicity problem, because overall the bookkeeping in *bal* is kept in sync with actual ether transfer, and with the builtin balance. In fact, the utility of the write *bal* := *b* in the function `withdraw` is different than that considered when adopting the criterion in [22]. This write is not for deducting an amount from the recorded balance to reflect ether transfer, but for canceling out the effect of an earlier deduction.

The second criterion is a security property called atomicity [16]. To see if a smart contract satisfies atomicity, one considers two executions of the contract, started in states that differ only in the gas value. If the two executions end with different values for some declared state variable(s), then it is required that the values of all the state variables should be the same as in the beginning, in at least one execution. Hence, the only allowed form of interference that the gas value has on the state is the suspension of changes to all state variables atomically. This atomicity property addresses a noninterference concern that is not addressed by our criterion. On the other hand, this property is not satisfied by the function `withdraw`. This is because if there is an insufficient amount of gas initially, causing the transfer in `withdraw` to fail, then the change to the value of *bal* is suspended, but the change to *err* is not. However, it is inappropriate to report this update of *err* as causing an atomicity problem, because this update is used to implement the logging functionality, and it does not cause the builtin balance of the contract to be out of sync with the bookkeeping balance.

Remark 1. In the motivating example, we use an exception-handling programming construct. Language features like this are introduced with Solidity version 0.6. Nevertheless, if the try ... catch ... were replaced by an if statement whose conditional checks whether a low-level call implementing the transfer succeeds or not, the essence of the example would remain.

3 Smart Contract Language

We consider a simplified programming abstraction for smart contracts. There is a blockchain system that contains a number of user accounts. Each account has

$$
\begin{aligned}
\theta &::= \mathsf{U160} \mid \mathsf{U256} \mid \mathsf{bool} \\
tp &::= \theta \mid \theta_1 \to \theta_2 \\
l &::= v : \theta \ \ (\text{where } v \in \mathbb{N} \cup \{tt, ff\}) \\
tvs &::= x_1 : \theta_1, \ldots, x_n : \theta_n \\
tpvs &::= x_1 : tp_1, \ldots, x_n : tp_n \\
e &::= l \mid x \mid x\langle e \rangle \mid e_1\ aop\ e_2 \mid e_1\ cop\ e_2 \mid e_1\ bop\ e_2 \mid \\
&\quad \mathsf{this} \mid \mathsf{caller} \mid \mathsf{callval} \mid \mathsf{balance} \mid \mathsf{adr}(c) \\
S &::= \mathsf{skip} \mid e_1 := e_2 \mid \mathsf{if}\ (e)\ \{S\} \mid \mathsf{while}\ (e)\ \{S\} \mid S_1; S_2 \mid \\
&\quad e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x \mid e_{\mathrm{a}}.\mathsf{transfer}(e) \mid \mathsf{throw} \mid \mathsf{try}\ \{S_1\}\ \mathsf{catch}\ \{S_2\} \\
fd &::= \mathsf{fun}\ f(tvs_1; tvs_2)\ \mathsf{ret}\ x : \theta\ \{S\} \\
fbd &::= \mathsf{fun}\ (tvs)\ \{S\} \\
ctr &::= \mathsf{contract}\ c\ \{\ \mathsf{var}\ tpvs\ fd_1\ \ldots\ fd_m\ fbd\ \} \\
ctrs &::= [ctr_1, \ldots, ctr_n]
\end{aligned}
$$

Fig. 2. The syntax of the smart contract language

an address, maintains its balance in the native digital currency of the blockchain, contains a storage that maps the state variables to their values, and contains a smart contract. Although we do not aim for a formalization of the computation model that is absolutely precise for Ethereum, we use "ether" to refer to the native digital currency, to aid the intuition of a reader familiar with Ethereum.

3.1 Syntax

We define a smart contract language with the syntax in Fig. 2. A *basic type* θ can be U160 that represents 160-bit addresses of user accounts, U256 that represents 256-bit unsigned integers, or the Boolean type bool. A *type tp* can be a basic type, or a mapping type $\theta_1 \to \theta_2$. A *literal* l can be a Boolean literal, or a numeral with value v and type θ. A *basic typed variable list tvs* is a list of variables each associated with a basic type. A *typed variable list tpvs* is a list of variables each associated with a type.

An *expression* e can be a literal, a variable reference, the expression $x\langle e \rangle$ that is used to retrieve the value for the key e in the mapping x or to update this value, an arithmetic operation, a comparative operation, a Boolean operation, the retrieval of the address of a contract ($\mathsf{adr}(c)$), or one of the operations retrieving state information about the current execution. The expression this retrieves the address of the currently executing contract. The expression caller retrieves the address of the caller contract (i.e., the contract calling the currently executing function). The expression callval retrieves the amount of ether transfered along with the call. The expression balance retrieves the ether balance of the currently executing contract.

A *statement* S can be an assignment, a branching construct, a looping construct, a sequential composition, or one of the constructs with strong association

to smart contract programming. The statement $e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x$ invokes the function with identifier f in the contract at the address e_{a}, passing n expressions as arguments, and putting the return value in the local variable x. The statement $e_{\mathrm{a}}.\mathsf{transfer}(e)$ transfers the amount e of ether to the address e_{a}. The statement throw throws an exception. The statement $\mathsf{try}\ \{S_1\}\ \mathsf{catch}\ \{S_2\}$ catches exceptions resulting from a call or a throw in S_1, and handles them in S_2.

A *function definition fd* contains a list tvs_1 of formal parameters, a list tvs_2 of local variables, a typed return variable, and a statement S that is the function body. A *fallback function* in a contract is executed when a call is made to a non-existing function in the contract, or after the contract receives ether. The definition of a fallback function *fbd* consists only of a list *tvs* of local variables and a function body. The fallback functions in our language resemble the fallback functions in Solidity [2].

A *contract ctr* consists of a contract identifier c, a typed variable list (with state variables), a list of function definitions, and a fallback function definition. Finally, *ctrs* models a list of contracts deployed together. A contract in *ctrs* may refer to another contract in the same list using the identifier of the latter.

We use *Tp* to represent the set of types, use L to represent the set of literals, use *Var* to represent the set of variables, use C to represent the set of contract identifiers, use *Ctr* to represent the set of contracts, and use *Fid* to represent the set of function identifiers. We adhere mostly to the rule that the initial letters for the names of sets are in upper case, and the initial letters for the meta-variables represents the individual elements are in lower-case. The only exception is the use of the meta-variable S for individual statements.

The language design reflects a number of key features of Ethereum smart contracts – the differentiation of local variables and state variables, mappings, ether transfer with post-processing at the receiver, exception handling, etc. Intra-contract and inter-contract calls are supported through the unified construct $e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x$. If the address of the callee equals that of the current contract, then an internal call is made; otherwise an external call is made.

3.2 Semantics

We present the key facts about the static semantics (i.e., type system) and dynamic semantics of our smart contract language. For space reasons, the complete technical development is given in the addendum of this paper.

Static Semantics The type system establishes the judgment

$$\vdash \mathit{ctrs}$$

This judgment says that the list *ctrs* of contracts is well-typed. The main requirements are that the contract identifiers should be pairwise distinct, and that each contract *ctr* in the list should be typable. The typability of each individual contract is captured by the judgment

$$\mathit{ctrs} \vdash \mathit{ctr}$$

In a contract *ctr*, the declaration of state variables with their types induces a *storage typing environment* in the set $Var \to Tp \cup \{\bot\}$. We denote this storage typing environment by $ste\text{-}of(ctr)$. In a function with the identifier $f_\bot \in Fid \cup \{\bot\}$ in the contract *ctr* ($f_\bot = \bot$ for the fallback function), the declaration of the formal parameters, local variables, and the return variable induces a *local typing environment* in the set $Var \to (L \cup \{\bot\})$. We denote this local typing environment by $lte\text{-}of(ctr, f_\bot)$. To establish $ctrs \vdash ctr$, the statements and expressions in the contract *ctr* are typed under $ste\text{-}of(ctr)$ and $lte\text{-}of(ctr, f_\bot)$.

Remark 2. A key reason for the contract *ctr* to be typed with a given list of contracts (*ctrs*) is the following. For a function call in the code of *ctr*, if the target contract of the call is specified by the contract identifier (i.e., $\mathsf{adr}(c).f(e_1, \ldots, e_n) \to x$), then the type system checks that c is in the set of contract identifiers of *ctrs*. Moreover, the existence of a function whose signature matches the call is statically checked in the target contract. Hence, the invocation of the fallback function in the target contract at runtime is avoided.

Dynamic Semantics We define a big-step semantics for our smart contract language. We introduce the semantic domains and the main judgment below.

Let $\mathbb{N}_k$ represent the subset of natural numbers up to $2^k - 1$. Let $\mathcal{D} := \{\mathbb{N}_{256}, \mathbb{N}_{160}, \{tt, ff\}\}$. Let $SVal := \bigcup \mathcal{D} \cup \bigcup_{A_1, A_2 \in \mathcal{D}} (A_1 \to A_2)$ be the set of *structured values*. We model the status of an account by an *account state* in the set $ASt := \mathbb{N}_{256} \times Ctr \times (Var \to SVal)$. For each $ast \in ASt$, *ast* is of the form (bal, ctr, st), where *bal* represents the balance of the account, *ctr* represents the smart contract of the account, and *st* represents the storage of the account. We model the state of the blockchain by *world states* in the set $WSt := \mathbb{N}_{160} \to ASt \cup \{\bot\}$. Each address value is mapped to an optional account state by a world state *wst*. We model the context for the current call by *execution environments* in the set $EE := \mathbb{N}_{160} \times \mathbb{N}_{160} \times \mathbb{N}_{256}$. For each $ee \in EE$, *ee* is of the form (ths, clr, cvl), where *ths* represents the address of the currently executing contract, *clr* represents the address of the caller contract, and *cvl* represents the amount of ether transfered with the current call. We record the values of the local variables of an executing function by local states in the set $LSt := Var \to (L \cup \{\bot\})$. Hence, each variable is mapped to an optional literal. That is, a local state *lst* records the types of the variables alongside their values. We model the deployment of contracts at addresses by *address environments* in the set $H := C \to \mathbb{N}_{160}$. This set is ranged over by η. We use the dot-notation to reference a component of a tuple. For instance, we refer to the balance of an account by $ast.bal$.

The evaluation of statements is represented by the judgment

$$\langle S, lst, wst, ee \rangle \to_\eta (lst', wst', cs)_{flg}$$

This judgment says: the evaluation of the statement S in the local state *lst*, the world state *wst*, the execution environment *ee*, and the address environment η results in the local state lst', the world state wst', the set *cs* of successful calls,

and the Boolean flag *flg* indicating the existence of uncaught exceptions. More concretely, *cs* is a subset of

$$\{call(v, v', f_\perp) \mid v, v' \in \mathbb{N}_{160} \wedge f_\perp \in Fid \cup \{\perp\}\}$$

Here, $call(v, v', f_\perp)$ represents a successful call from the account at the address v to the account at the address v'. The $f_\perp$ is an optional function identifier for the intended callee. In case $f_\perp = \perp$, a transfer is intended.

3.3 Preservation of Types by Evaluation

We show that types and the observance of types by the values of variables are preserved by evaluation.

We write $ste \blacktriangleright st$ to express that the storage type environment *ste* is consistent with the storage *st*. Intuitively, each state variable is mapped under *st* to a value (which could be a function value) in the range of its type according to *ste*. We write $lte \triangleright lst$ to express that the local type environment *lte* is consistent with the local state *lst*. Intuitively, each local variable is mapped under *lst* to a value in the range of its type according to *lte*.

We write $wf(ctr, f_\perp, lst, wst)$ to express that the values of all local variables, parameters and the return variable of the function identified by $f_\perp$ in the contract *ctr* as recorded in *lst*, and the values of all state variables (of any contract) as recorded in *wst*, are consistent with the types of these variables, and that each contract in *wst* is well-typed under some list *fds* of contracts.

$$\begin{array}{l} wf(ctr, f_\perp, lst, wst) := \\ \quad lte\text{-}of(ctr, f_\perp) \triangleright lst \wedge \\ \quad (\forall v \in \mathbb{N}_{160} : \forall ast : wst(v) = ast \Rightarrow ste\text{-}of(ast.ctr) \blacktriangleright ast.st \wedge \exists ctrs : ctrs \vdash ast.ctr) \end{array}$$

We then have the following result.

Theorem 1. *If $wf(ctr, f_\perp, lst, wst)$ holds, $ctr = wst(ee.ths).ctr$ holds, $S = stmt\text{-}of(ctr, f_\perp)$ holds, and $\langle S, lst, wst, ee\rangle \rightarrow_\eta (lst', wst', cs)_{flg}$ can be derived, then $wf(ctr, f_\perp, lst', wst')$ holds.*

Here, $stmt\text{-}of(ctr, f_\perp)$ gives the body of the function identified by $f_\perp$ in the contract *ctr*. The theorem indicates that if a well-typed function is executed to completion, and the values of local and state variables observe their types initially, then the local and state variables still have the same types that are observed by their values in the end.

4 Program Logic

The program logic for our smart contract language establishes Hoare-style judgments for the functions of smart contracts. It explicitly records the presence of successful calls via dedicated terms, and implicitly keeps track of the presence of state updates via logical variables. In this manner, it supports the reasoning about whether specific calls and state updates are performed in atomic batches.

4.1 The Assertions

We consider the following language for formulating the assertions in the program logic. These assertions are used as the pre-conditions and post-conditions for statements and functions, as well as the invariants for contracts.

$$\begin{aligned} \underline{t} &::= \nu \mid w \mid x \mid x^c \mid \mathfrak{b} \mid \kappa \mid u \mid \mathit{ths} \mid \mathit{clr} \mid \mathit{cvl} \mid \varepsilon \\ t &::= \underline{t} \mid \overline{c} \mid t_1 \mathit{aop}\, t_2 \mid \underline{t}(t) \mid t[t_1 : t_2] \mid (t_1, t_2, f_\perp) \\ \phi &::= t_1 \mathit{cop}\, t_2 \mid \phi_1 \mathit{bop}\, \phi_2 \mid \exists \underline{t} : \phi \end{aligned}$$

A *basic term* $\underline{t}$ can be a value[4] (ν), an auxiliary variable (w), a local variable (x), a state variable annotated with the identifier of its contract (x^c), the variable $\mathfrak{b}$ representing a mapping from account addresses to balances, the variable κ representing a mapping from triples $(\nu_1, \nu_2, f_\perp)$ to Boolean values for the presence of successful calls from the account at address ν_1 to the account at address ν_2, targeting the function identified by $f_\perp$, a logical variable u representing the value of a program variable, $\mathfrak{b}$ or κ, at a fixed program point, *ths* representing the address of the currently executing contract, *clr* representing the address of the caller, *cvl* representing the call value, or ε signalling an uncaught exception.

A *term* t can be a basic term, the term $\overline{c}$ for the address of the contract with identifier c, an arithmetic operation on terms, $\underline{t}(t)$ for the application of $\underline{t}$ on t, $t[t_1 : t_2]$ for the update of t (that represents a function) at point t_1 to t_2, or a triple $(t_1, t_2, f_\perp)$ consisting of a source contract address, a destination contract address, and an optional function identifier for a call. Whether such a call has been successfully performed is represented by $\kappa(t_1, t_2, f_\perp)$.

An *assertion* ϕ can be a comparison of two terms, a Boolean operation on two assertions, or an assertion with a quantified basic term.

We use ε as syntactical sugar for $\varepsilon = \mathit{tt}$, and use $\neg\varepsilon$ as syntactical sugar for $\varepsilon = \mathit{ff}$. For an assertion ϕ, we use $\neg\phi$ as syntactical sugar for $((0 = 0)\, \mathit{xor}\, \phi)$, and we use $\forall \underline{t} : \phi$ as syntactical sugar for $\neg(\exists \underline{t} : \neg\phi)$.

We write $\mathit{lvs}(\phi)$ for the set of logical variables in the assertion ϕ. We write $\mathit{svs}(a_1, \ldots, a_n)$ for the set of state variables (of the form x^c for some x and c) in any of a_1, …, a_n. We write $\mathit{vs}(a_1, \ldots, a_n)$ for the set of variables in any of a_1, …, a_n. Here, each a_i is an assertion or a term. We precede a set of variables with $\exists$ to represent a series of existential quantifications over the variables in this set. We write $\phi[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]$ for the simultaneous substitution of t_1, …, t_n for $\underline{t}_1$, …, $\underline{t}_n$, in ϕ, where $\underline{t}_1$, …, $\underline{t}_n$ are pair-wise distinct.

We write $[e]^c_X$ for the term or assertion corresponding to the expression e. Here, c is the identifier of the contract in which e resides, and X is the set of local variables of the function in which e resides. We define $[x]^c_X := x$ if $x \in X$, and $[x]^c_X := x^c$ if $x \notin X$. We define $[x\langle e\rangle]^c_X := x^c([e]^c_X)$, where the right-hand side represents the application of x^c on $[e]^c_X$. The definition of $[e]^c_X$ on other expressions are relatively straightforward.

[4] We allow function values in the assertions, and, hence, ν is used rather than v.

4.2 The Inference System

We write Δ to represent a function that takes each contract identifier c and function identifier f to a pair $([x_1, \ldots, x_n, x], (\Phi, \Phi'))$. Here, x_1, $\ldots$, x_n are the formal parameters of the function identified by f in c, and x is the return variable of this function. Furthermore, Φ and Φ' are the *pre-condition* and *post-condition* of this function, respectively. We restrict the local variables of a function that are used in its pre-condition to the formal parameters. We restrict the local variables of a function that are used in its post-condition to the return variable.

The *invariant* I for a contract satisfies: If I holds before any function of the contract is called, then I holds after the function finishes executing. Furthermore, we assume that the only basic terms contained in the invariant for a contract with the contract identifier c are the state variables of the contract, logical variables, $\mathfrak{b}$, and κ, where $\mathfrak{b}$ is only used in the term $\mathfrak{b}(\overline{c})$, and κ is only used in terms $\kappa(\overline{c}, t, f_\perp)$ for some t and $f_\perp$. Hence, after the contract with identifier c makes a call, the values for the terms involving $\mathfrak{b}$ and κ in I can only change when the contract is re-entered.

The logical judgment for statements is

$$I, \Delta \vdash^{c}_{X} \{\phi\}\ S\ \{\phi'\}$$

The judgment says that under I and Δ, if the pre-condition ϕ holds when starting to evaluate the statement S, and the evaluation terminates, then the post-condition ϕ' holds on termination. The statement S is part of the contract with identifier c, with accesses to local variables in X.

The inference rules for statements that are neither calls nor transfers are shown in Fig. 3. The two rules for assignments are formulated to support forward reasoning [14]. For the rule about assignments to variables x, the post-condition says that there exists some initial value w for x, such that the pre-condition holds for this value, and an equality holds between the two sides of the assignment, provided that the expression e is evaluated using the value of w for x. The premise of this rule requires that w should be fresh for ϕ. The rule for assignments to mappings embodies similar intuition. The rule for sequential compositions $S_1; S_2$ derives a post-condition that reflects the post-state can either be an exceptional post-state of S_1, or the post-state of S_2 in case S_1 finishes normally. The rule for $\mathsf{try}\ \{S_1\}\ \mathsf{catch}\ \{S_2\}$ employs the pre-condition $\phi''[tt/\varepsilon] \wedge \neg\varepsilon$ for S_2, where ϕ'' is the post-condition for S_1. The substitution erases the information about the exception from S_1 because the exception is caught. The conjunction with $\neg\varepsilon$ signals to S_2 that there is no current exception. The remaining rules in Fig. 3 can be understood following intuition from standard Hoare logic [4].

The inference rules for call statements are presented in Fig. 4. The first rule describes the case where the identifier (c') of the target contract is known. In the premise, it is checked that the pre-condition Φ of the callee function should hold with substitutions of the argument expressions for the formal parameters, as well as the substitutions in δ, under the pre-condition ϕ of the call. Here, the substitutions in δ take the execution environment used for the evaluation of Φ

$$\frac{w \notin vs(\phi)}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ x := e\ \{\exists w : \phi[w/[x]^{\mathrm{c}}_X] \wedge [x]^{\mathrm{c}}_X = [e]^{\mathrm{c}}_X[w/[x]^{\mathrm{c}}_X] \wedge \neg\varepsilon\}}$$

$$\frac{w \notin vs(\phi) \quad \phi' = \exists w : \phi[w/x^{\mathrm{c}}] \wedge x^{\mathrm{c}} = w[[e_1]^{\mathrm{c}}_X[w/x^{\mathrm{c}}] : [e_2]^{\mathrm{c}}_X[w/x^{\mathrm{c}}]] \wedge \neg\varepsilon}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ x\langle e_1\rangle := e_2\ \{\phi'\}}$$

$$\frac{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ S_1\ \{\phi''\} \qquad I, \Delta \vdash^{\mathrm{c}}_X \{\phi'' \wedge \neg\varepsilon\}\ S_2\ \{\phi'\}}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ S_1; S_2\ \{\phi'' \wedge \varepsilon \vee \phi'\}}$$

$$\frac{I, \Delta \vdash^{\mathrm{c}}_X \{\phi \wedge [e]^{\mathrm{c}}_X\}\ S\ \{\phi'\}}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ \mathsf{if}\ (e)\ \{S\}\ \{\phi' \vee \phi \wedge \neg[e]^{\mathrm{c}}_X \wedge \neg\varepsilon\}}$$

$$\frac{I, \Delta \vdash^{\mathrm{c}}_X \{\phi \wedge [e]^{\mathrm{c}}_X\}\ S\ \{\phi\}}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ \mathsf{while}\ (e)\ \{S\}\ \{\phi \wedge \neg[e]^{\mathrm{c}}_X \wedge \neg\varepsilon \vee \phi \wedge \varepsilon\}}$$

$$\frac{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ S_1\ \{\phi''\} \qquad I, \Delta \vdash^{\mathrm{c}}_X \{\phi''[tt/\varepsilon] \wedge \neg\varepsilon\}\ S_2\ \{\phi'\}}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ \mathsf{try}\ \{S_1\}\ \mathsf{catch}\ \{S_2\}\ \{\phi'' \wedge \neg\varepsilon \vee \phi'\}}$$

$$\frac{}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ \mathsf{skip}\ \{\phi\}} \qquad \frac{}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ \mathsf{throw}\ \{\phi[\mathit{ff}/\varepsilon] \wedge \varepsilon\}}$$

$$\frac{I, \Delta \vdash^{\mathrm{c}}_X \{\phi'_1\}\ S\ \{\phi'_2\} \quad \phi_1 \Rightarrow \phi'_1 \quad \phi'_2 \Rightarrow \phi_2}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi_1\}\ S\ \{\phi_2\}}$$

Fig. 3. The inference rules for statements (part 1)

to that of the callee. In the conclusion of this rule, the condition ϕ' in the post-condition describes the states reached if the call succeeds. In ϕ', it is stated that there is some return value w, and some set of successful calls described by the function w', for which the post-condition Φ' holds in the execution environment of the callee. In addition, the final mapping κ for the successful calls is as w', except that the success of the current call from $\overline{c}$ to $\overline{c'}$ targeting f is also recorded. The additional conditions in ϕ' state that the pre-conditions ϕ and Φ hold in the post-state, if the updated variables are re-mapped to some appropriate values (e.g., their values in the pre-state). The inclusion of these conditions enables information to be passed directly, or via the logical variables shared by Φ and Φ', from the pre-state to the post-state. The second rule in Fig. 4 describes how to reason about a call for which the identifier of the target contract is unavailable. In the post-condition, ϕ' describes the post-states reached after the call succeeds. The first disjunct of ϕ' describes post-states reached without any successful reentrance to the current contract. It reflects the fact that the state variables of the current contract still have their initial values after the call, and the mapping of successful calls is related to the initial mapping w' as $\kappa \overset{\mathrm{c},1}{=} w'[[e_{\mathrm{a}}]^{\mathrm{c}}_X[w/\mathfrak{b}], f : tt] \wedge \kappa \overset{\mathrm{c},2}{=} w'$. The two conjuncts of this condition are defined according to Fig. 5. Hence, the condition $\kappa \overset{\mathrm{c},1}{=} w'[[e_{\mathrm{a}}]^{\mathrm{c}}_X[w/\mathfrak{b}], f : tt] \wedge \kappa \overset{\mathrm{c},2}{=} w'$ says that there is a successful call from the current contract to the callee, and otherwise the successful calls from the current contract, or to

$$\frac{\begin{array}{c}\Delta(c', f) = ([x_1, \ldots, x_n, x'], (\Phi, \Phi')) \qquad \delta = [\overline{c'}/ths][\overline{c}/clr][0/cvl] \\ \phi \wedge \neg\varepsilon \;\Rightarrow\; \exists lvs(\Phi) : \Phi[[e_1]^{\mathrm{c}}_X/x_1, \ldots, [e_n]^{\mathrm{c}}_X/x_n]\delta \quad vs(\Phi') \cap \{w, w'\} = \emptyset \\ lvs(\phi) \cap lvs(\Phi) = \emptyset \qquad (\bigcup_i vs([e_i]^{\mathrm{c}}_X)) \cap \{ths, clr, cvl\} = \emptyset\end{array}}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ \mathsf{adr}(c').f(e_1, \ldots, e_n) \to x\ \{\phi' \wedge \neg\varepsilon \vee \phi[ff/\varepsilon] \wedge \varepsilon\}}$$

where $\phi' = \exists lvs(\Phi) :$

$$\left(\begin{array}{l}\exists \mathfrak{b} : \exists \kappa : \exists svs(\phi, \Phi, [e_1]^{\mathrm{c}}_X, \ldots, [e_n]^{\mathrm{c}}_X) : \exists x : \\ \quad \phi \wedge \Phi[[e_1]^{\mathrm{c}}_X/x_1, \ldots, [e_n]^{\mathrm{c}}_X/x_n]\delta\end{array}\right)$$
$$\wedge\, \exists w, w' : \Phi'[w/x'][w'/\kappa]\delta \wedge x = w \wedge \kappa = w'[(\overline{c}, \overline{c'}, f) : tt]$$

$$\frac{lvs(\phi) \cap lvs(I) = \emptyset \quad \phi \wedge \neg\varepsilon \Rightarrow \exists lvs(I) : I \quad vs(\phi) \cap \{w, w'\} = \emptyset}{I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x\ \{\phi' \wedge \neg\varepsilon \vee \phi[ff/\varepsilon] \wedge \varepsilon\}}$$

where $\phi' =$

$$\left(\begin{array}{l}\exists w : \exists w' : \exists(svs(\phi) \setminus \{x^c \mid x \in Var\}) : \exists x : \\ \quad \phi[w/\mathfrak{b}][w'/\kappa] \;\wedge\; \mathfrak{b}(\overline{c}) = w(\overline{c}) \wedge \\ \quad \kappa \overset{\mathrm{c},1}{=} w'[[e_{\mathrm{a}}]^{\mathrm{c}}_X[w/\mathfrak{b}], f : tt] \;\wedge\; \kappa \overset{\mathrm{c},2}{=} w'\end{array}\right) \vee$$
$$\exists lvs(I) : \exists w :$$
$$\left(\begin{array}{l}(\exists svs(\phi, I, [e_{\mathrm{a}}]^{\mathrm{c}}_X) : \exists \mathfrak{b} : \exists x : (\exists \kappa : \phi \wedge I) \;\wedge\; w = [e_{\mathrm{a}}]^{\mathrm{c}}_X) \\ \wedge\, \exists w' : I[w'/\kappa] \wedge \kappa = w'[(\overline{c}, w, f) : tt]\end{array}\right)$$

Fig. 4. The inference rules for statements (part 2)

$$\kappa \overset{\mathrm{c},1}{=} w'[t, f_\perp : tt] \;:=$$
$$\left(\begin{array}{l}\kappa(\overline{c}, t, f_\perp) = tt \wedge \\ \forall w'', f'_\perp : (w'' \neq t \vee f'_\perp \neq f_\perp) \Rightarrow \kappa(\overline{c}, w'', f'_\perp) = w'(\overline{c}, w'', f'_\perp)\end{array}\right)$$
$$\kappa \overset{\mathrm{c},2}{=} w' \;:=\; (\forall w'', f'_\perp : \kappa(w'', \overline{c}, f'_\perp) = w'(w'', \overline{c}, f'_\perp))$$

Fig. 5. The definitions of $\kappa \overset{\mathrm{c},1}{=} w'[t, f_\perp : tt]$ and $\kappa \overset{\mathrm{c},2}{=} w'$

the current contract, are the same as before. The second disjunct of ϕ' describes the post-states reached after successful reentrance to the current contract during the call. The invariant I of the current contract is used to establish the post-condition. It is stated that there is some function w' describing the presence of all successful calls immediately before the return of the current call, such that I is satisfied for w', and the new set of successful calls is as described by w', except that the call from the current contract to the target contract should be added (last line of the second disjunct).

The inference rules for transfer statements are presented in Fig. 6. The rules for transfers come with a case distinction in whether the identifier of the target contract is explicitly specified in the transfer statement. If the identifier of the target contract is explicitly specified, the pre-condition and post-condition of the callee function (i.e., the fallback function of the target contract) are leveraged in reasoning about the transfer. Otherwise, the invariant of the current contract is used. In the premise of the rule for the transfer statement $\mathsf{adr}(c').\mathsf{transfer}(e)$, it is checked that the precondition of the callee function is implied by the fact

$$\frac{\begin{array}{c}\Delta(c',f) = ([],(\Phi,\Phi')) \quad \delta = [\overline{c'}/ths][\overline{c}/clr] \quad lvs(\phi) \cap lvs(\Phi) = \emptyset \\ \left(\begin{array}{l} c \neq c' \wedge (\phi \wedge \neg\varepsilon) \wedge [e]^{\mathsf{c}}_X \leq \mathfrak{b}(\overline{c}) \wedge \mathfrak{b}(\overline{c'}) + [e]^{\mathsf{c}}_X < 2^{256} \Rightarrow \\ \exists lvs(\Phi) : \Phi[\mathfrak{b}[\overline{c} : \mathfrak{b}(\overline{c}) - [e]^{\mathsf{c}}_X, \overline{c'} : \mathfrak{b}(\overline{c'}) + [e]^{\mathsf{c}}_X]/\mathfrak{b}]\delta[[e]^{\mathsf{c}}_X/cvl] \end{array}\right) \\ vs([e]^{\mathsf{c}}_X) \cap \{ths, clr, cvl\} = \emptyset \quad vs(\phi,\Phi,\Phi') \cap \{w, w', w_0, w_1\} = \emptyset \end{array}}{I, \Delta \vdash^{\mathsf{c}}_X \{\phi\}\ \mathsf{adr}(c').\mathsf{transfer}(e)\ \{\phi' \wedge \neg\varepsilon \vee \phi[\mathit{ff}/\varepsilon] \wedge \varepsilon\}}$$

where $\phi' = \exists lvs(\Phi) : \exists w :$

$$\left(\begin{array}{l} \exists w_0 : \exists w_1 : \exists \kappa : \exists svs(\phi, \Phi, [e]^{\mathsf{c}}_X) : \\ \quad w = [e]^{\mathsf{c}}_X[w_0/\mathfrak{b}] \wedge \phi[w_0/\mathfrak{b}] \wedge \Phi[w_1/\mathfrak{b}]\delta[w/cvl] \wedge \\ \quad w_1 = w_0[\overline{c} : w_0(\overline{c}) - w, \overline{c'} : w_0(\overline{c'}) + w] \end{array}\right)$$

$$\wedge\ \exists w' : \Phi'[w'/\kappa]\delta[w/cvl]\ \wedge\ \kappa = w'[(\overline{c}, \overline{c'}, \bot) : tt]\ \wedge\ c \neq c'$$

$$\frac{\begin{array}{c} lvs(\phi) \cap lvs(I) = \emptyset \qquad vs(\phi, I) \cap \{w, w', w_0, w_1\} = \emptyset \\ \left(\begin{array}{l} \overline{c} \neq [e_{\mathrm{a}}]^{\mathsf{c}}_X \wedge (\phi \wedge \neg\varepsilon) \wedge [e]^{\mathsf{c}}_X \leq \mathfrak{b}(\overline{c}) \wedge \mathfrak{b}([e_{\mathrm{a}}]^{\mathsf{c}}_X) + [e]^{\mathsf{c}}_X < 2^{256} \\ \Rightarrow \exists lvs(I) : I[\mathfrak{b}[\overline{c} : \mathfrak{b}(\overline{c}) - [e]^{\mathsf{c}}_X, [e_{\mathrm{a}}]^{\mathsf{c}}_X : \mathfrak{b}([e_{\mathrm{a}}]^{\mathsf{c}}_X) + [e]^{\mathsf{c}}_X]/\mathfrak{b}] \end{array}\right) \end{array}}{I, \Delta \vdash^{\mathsf{c}}_X \{\phi\}\ e_{\mathrm{a}}.\mathsf{transfer}(e)\ \{\phi' \wedge \neg\varepsilon \vee \phi[\mathit{ff}/\varepsilon] \wedge \varepsilon\}}$$

where $\phi' =$

$$\left(\begin{array}{l} \exists w : \exists w' : \exists (svs(\phi) \setminus \{x^c \mid x \in Var\}) : \\ \quad \phi[w/\mathfrak{b}][w'/\kappa] \wedge \overline{c} \neq [e_{\mathrm{a}}]^{\mathsf{c}}_X[w/\mathfrak{b}] \wedge \mathfrak{b}(\overline{c}) = w(\overline{c}) - [e]^{\mathsf{c}}_X[w/\mathfrak{b}] \wedge \\ \quad \kappa \stackrel{c,1}{=} w'[[e_{\mathrm{a}}]^{\mathsf{c}}_X[w/\mathfrak{b}], \bot : tt]\ \wedge\ \kappa \stackrel{c,2}{=} w' \end{array}\right) \vee$$

$$\exists lvs(I) : \exists w :$$

$$\left(\begin{array}{l} \exists w_0 : \exists w_1 : \exists svs(\phi, I, [e_{\mathrm{a}}]^{\mathsf{c}}_X, [e]^{\mathsf{c}}_X) : \\ \quad (\exists \kappa : \phi[w_0/\mathfrak{b}] \wedge I[w_1/\mathfrak{b}])\ \wedge\ w = [e_{\mathrm{a}}]^{\mathsf{c}}_X[w_0/\mathfrak{b}]\ \wedge\ \overline{c} \neq w \wedge \\ \quad w_1 = w_0[\overline{c} : w_0(\overline{c}) - [e]^{\mathsf{c}}_X[w_0/\mathfrak{b}], w : w_0(w) + [e]^{\mathsf{c}}_X[w_0/\mathfrak{b}]] \end{array}\right)$$

$$\wedge\ \exists w' : I[w'/\kappa] \wedge \kappa = w'[(\overline{c}, w, \bot) : tt]$$

Fig. 6. The inference rules for statements (part 3)

that the transfer is performed towards a different contract than the current one, the precondition of the transfer, and the fact that the transfered amount can be supplied by the caller and does not cause any overflow of the callee's balance. In the conclusion of this rule, the condition ϕ' describes the states reached in case the transfer succeeds. In ϕ', it is stated that there exists some call value w (i.e., the amount of ether transfered to the callee), and some mapping w' for the successful calls that have been performed on completion of the callee function, such that the post-condition Φ' of the callee holds. The successful calls that have been performed after the returning of the callee are as those after the completion of the callee, except that the current transfer has also succeeded. It is also stated that the identifier of the current contract is different than the identifier of the target contract (in case the transfer succeeds). The remaining part of ϕ' says that there exists mappings w_0 and w_1 for the account balances immediately before the transfer is performed, and immediately before the execution of the callee function, respectively, such that the pre-condition ϕ of the transfer holds if the balances of the accounts are in accordance with w_0, and the pre-condition Φ of the callee function holds if the balances of the accounts are in accordance with

w_1. The rule for the transfer statement $e_{\mathrm{a}}.\mathsf{transfer}(e)$ is analogous to the rule for the transfer statement $\mathsf{adr}(c').\mathsf{transfer}(e)$ in the treatment of account balances. In addition, the rule for the transfer statement $e_{\mathrm{a}}.\mathsf{transfer}(e)$ is analogous to the rule for the call statement $e_{\mathrm{a}}.f(e_1,\ldots,e_n) \to x$ in the treatment of the two different cases regarding the presence of reentrance into the current contract.

Based on the verification of statements, functions and fallback functions can be verified against their specifications. Furthermore, a given contract *ctr* can be verified against the specifications of its functions in Δ, using an invariant I for the contract *ctr*. This gives rise to an instance of the judgment $I, \Delta \vdash ctr$. Ultimately, a given set of smart contracts can be verified against the specifications of their functions in Δ. This results in an instance of the judgment $\Delta \vdash ctrs$.

The program logic devised in the above lays a solid foundation for the verification of safety properties for smart contracts. Some of the logic rules have relatively involved formulations. To overcome the tediousness of using these rules in the reasoning tasks, a semi-automated tool can be implemented to conduct verification based on the specification of contract invariants and loop invariants.

4.3 Soundness

The soundness of our program logic relies on a notion of satisfaction of assertions in states. We interpret the assertions in *assertion states* of the form $\sigma{:=}(lst, wst, ee, cs, \mathit{flg}, \zeta)$. The first five components are all from the semantic judgment for statements. The last component, ζ, is a function that gives the values of the auxiliary variables w, as well as the logical variables u in the assertions. The interpretation result is written $[\![\phi]\!]_{\mathrm{asst}}(\sigma)$, which is a truth value.

We articulate the notion that a function semantically satisfies its specification that consists of a pre-condition and a post-condition. More concretely, we write $ctrs \models \{\Phi\}\ (ctr, f_\perp)\ \{\Phi'\}$ to express that under the following five conditions

1. $wf(ctr, f_\perp, lst, wst)$,
2. $\forall x \in \mathit{nprms\text{-}of}(ctr, f_\perp) : \exists \theta : lst(x) = d(\theta) : \theta$,
3. $\eta(cid(ctr)) = ee.ths \wedge \forall i : wst(\eta(cid(ctrs!i))).ctr = ctrs!i$,
4. $\langle \mathit{stmt\text{-}of}(ctr, f_\perp), lst, wst, ee\rangle \to_\eta (lst', wst', cs)_{\mathit{ff}}$,
5. $[\![\Phi]\!]_{\mathrm{asst}}(lst, wst, ee, cs_0, \mathit{ff}, \zeta) = tt$,

it holds that $[\![\Phi']\!]_{\mathrm{asst}}(lst', wst', ee, cs_0 \cup cs, \mathit{ff}, \zeta) = tt$.

Intuitively, the main requirement of $ctrs \models \{\Phi\}\ (ctr, f_\perp)\ \{\Phi'\}$ is that if the execution of the function with the identifier $f_\perp$ in the smart contract *ctr* starts in a state that satisfies the condition Φ (Condition 5), and the execution finishes without uncaught exceptions (Condition 4), then the ending state satisfies the condition Φ'. Condition 1 requires that the execution of the function should be started in local and world states in which the values of variables are consistent with their types (see Section 3.3). Condition 2 requires that the initial local state should map each local variable to the default value for its (basic) type, unless the variable is a parameter. Here, the default value for the basic type θ is written $d(\theta)$, which is defined as 0 for the basic types $\mathsf{U160}$ and $\mathsf{U256}$, and defined as *ff* for the basic type bool. Condition 3 is a sanity condition on the

address environment η used in the evaluation. It requires that the identifier of the currently executing contract should be mapped to the address of the current contract ($ee.ths$). Additionally, it requires that the identifier of each given contract should be mapped to an account address where the code of the contract can be found.

We have the following theorem about the soundness of the program logic.

Theorem 2. *If* $\vdash ctrs$ *and* $\Delta \vdash ctrs$ *can be derived,* ctr *is in* $ctrs$*, and* $\Delta(ctr, f_\perp) = (xs, (\Phi, \Phi'))$ *for some* xs*, then we have* $ctrs \models \{\Phi\}\ (ctr, f_\perp)\ \{\Phi'\}$.

Hence, if a given series of smart contracts are well-typed, and these contracts pass the verification using the program logic, then each function (or fallback function) of each given contract semantically satisfies its specification.

With the assertion language in Section 4.1, Theorem 2 supports the sound deductive verification of general safety requirements for smart contracts at the source-level. When logical variables are used in Φ and Φ' to help express the presence of semantic updates to state variables, and the term κ is used to help express the presence of successful calls, the sound reasoning about the satisfaction of the atomicity criterion proposed in Section 2 is supported (see Section 5).

5 Atomicity Verification

Firstly, we evaluate our technique using the example in Fig. 1 that was intuitively discussed to motivate our work. Secondly, we consider a more involved example with atomicity requirements verified on a function refunding a group of users.

Verifying Atomicity for Ether Withdrawal Suppose a smart contract ctr has identifier c, and it consists of the function `withdraw` of Fig. 1, and a fallback function with the code $\mathsf{if}\ (bal = 0)\ \{\mathsf{throw}\}$.

Suppose Δ has the domain $\{(c, \texttt{withdraw}), (c, \perp)\}$. Furthermore, suppose

$$\Delta(c, \texttt{withdraw}) := (\,[\,],\ (bal^c = u_1 \wedge \mathfrak{b} = u_2,\ bal^c \neq u_1 \Leftrightarrow \mathfrak{b}(\overline{c}) \neq u_2(\overline{c}))\,)$$
$$\Delta(c, \perp) := (\,[\,],\ (\mathit{ff}, \mathit{tt})\,)$$

Then, it can be established that $\Delta \vdash [ctr]$, using the invariant

$$I = (bal^c = 0 \wedge \mathfrak{b}(\overline{c}) = u)$$

for the contract ctr. This invariant states that if the state variable bal has value 0 before calling either function of the contract ctr, then bal still has the value 0, and the builtin balance of the contract remains the same (via the logical variable u), after the callee function finishes executing. If bal and err are both declared with the type $\mathsf{U256}$ in the contract ctr, then we also have $\vdash [ctr]$. This enables the establishment of the following result using Theorem 2.

$$[ctr] \models \{bal^c = u_1 \wedge \mathfrak{b} = u_2\}\ (ctr, \texttt{withdraw})\ \{bal^c \neq u_1 \Leftrightarrow \mathfrak{b}(\overline{c}) \neq u_2(\overline{c})\}$$

This result indicates that if we start the execution of the function `withdraw` in a legal initial state (i.e., with the values of b and x observing the type $\mathsf{U256}$, and the value of b being 0), and the execution finishes normally, then bal is

```
fun refund( ; i:U256) ret x:U256 {
  if (lock != 0) { throw };
  lock := 1; i := 0;
  while (i < num) {
    if (bals⟨addr⟨i⟩⟩ > 0) {
      try { addr⟨i⟩.transfer(bals⟨addr⟨i⟩⟩); bals⟨addr⟨i⟩⟩ := 0 }
      catch { skip }
    };
    i := i + 1
  };
  lock := 0
}
```

Fig. 7. The function `refund`

semantically updated if and only if the builtin balance of the contract is updated. If the execution ends exceptionally, then there is no effect on the states. This means that the fact established in the above applies for all possible executions started from legal initial states for the function `withdraw`. These include both the executions that end normally, and the executions that end exceptionally.

Thus, it is verified that the function `withdraw` always performs the actual transfer and the bookkeeping in an atomic batch. □

Verifying Per-account Atomicity for Batch Transfer We consider a more involved example, where a contract *ctr* with identifier c consists of the function `refund` in Fig. 7, and a fallback function with code if $(lock$!= $0)$ {throw}.

Via a `while` loop, the function `refund` transfers the amount $bals\langle addr\langle i\rangle\rangle$ of ether to the user account at address $addr\langle i\rangle$, for each i from 0 to $num - 1$. To avoid reentrance, the state variable *lock* is used. Each time a transfer is made, *lock* has the value 1. Hence, an exception is thrown if the receiver attempts at a call back to `refund` or the fallback function of the contract *ctr*.

The rationale of the function `refund` is to refund as many users as possible in a single execution. The transfer to a specific user may fail, without affecting the transfer to a different user. Hence, it cannot be guaranteed in general that all the transfers happen in an atomic batch. Nevertheless, it is crucial to ensure that for each individual user, the transfer and the setting of the bookkeeping balance to 0 (with $bals\langle addr\langle i\rangle\rangle := 0$) must happen together.

This atomicity requirement can be verified using the proposed program logic. With the pre-condition

$$\forall r : 0 \leq r < num^c \Rightarrow (bals^c(addr^c(r)) = u(r) \wedge \kappa(\overline{c}, addr^c(r), \bot) = \mathit{ff})$$

for the function `refund`, the post-condition

$$\forall r : 0 \leq r < num^c \Rightarrow (bals^c(addr^c(r)) \neq u(r) \Leftrightarrow \kappa(\overline{c}, addr^c(r), \bot) = \mathit{tt})$$

can be derived. This indicates that the bookkeeping balance for the r-th user is semantically updated, if and only if a transfer to the r-th user succeeds. □

The two examples given in this section are both about atomicity requirements on a single smart contract, in a scenario involving ether transfer. In general, the

usage scope of the proposed program logic is not limited to the verification of requirements on a single smart contract, or ether-related scenarios. This is because pre-conditions and post-conditions involving any state variables of multiple smart contracts that are deployed together can be specified. For instance, in a retailing scenario, if the information about the products and their current stock are managed in two different smart contracts, the proposed program logic can be used to verify that the registration of a product in one of the smart contracts must happen together with the initialization of its stock in the other. Last but not least, the usage scope of the proposed program logic goes beyond the verification of atomicity requirements. This is because the Hoare-style pre-conditions and post-conditions support the verification of partial correctness in general.

6 Related Work

Design and Formalization of Smart Contract Languages To support the sound verification of smart contracts, there have been a number of developments on the formalization of smart contract languages.

In [13], a calculus called Featherweight Solidity is defined to closely model the core of the Solidity language. The calculus supports contract creation and single inheritance, which are not supported by our language. In [6], a minimal calculus for Solidity contracts is formalized. The formalization contains an explicit model of transactions and blockchains, while our formalization is focused on the execution of a single transaction. Neither work deals with the verification of smart contracts beyond type checking. On the other hand, we provide a discussion of atomicity-related requirements on smart contracts, and devise a program logic for the source-level verification of smart contracts.

In [34] and [21], formalizations of large fragments of Solidity are provided, with a big-step semantics, and a small-step semantics, respectively. Both formalizations are mechanized – the first in the Coq proof assistant, and the second in the K Framework. In [19, 16, 18], the bytecode language of the Ethereum virtual machine is formalized. These works are mostly focused on the semantic foundation for smart contracts, without looking much into verification problems.

The work [28] does not formalize an existing smart contract language, but proposes a new language for the safe programming of smart contracts. This is an intermediate language with strong safety guarantees provided by its type system. Light-weight static analyses are defined to address some of the verification issues. The design of a new language simplifies the verification of smart contracts.

Formal Validation of Smart Contracts There have been extensive research efforts into the formal validation of smart contracts. The consideration of soundness is featured by a small fragment of the existing developments.

In [26], a technique based on software model checking is proposed for the automated verification of Ethereum smart contracts. The approach is sound in general. However, there is no mentioning of a formal semantics on which soundness arguments are based. In [17], a tool for the source-level verification of Solidity contracts is presented. Semi-automated deductive verification is performed, but the verification does not appear to be based on solid semantic foundation.

Several developments exist on the analysis and verification of Ethereum smart contracts in low-level and intermediate-level languages. In [15] and [27], static analysis techniques and tools for the sound checking of EVM bytecode are presented. In [22], a verification technique for the LLVM intermediate representation of smart contracts is developed. Our development differs from these existing ones both in the language level targeted, and in the verification approach taken.

Finally, in [3], a program logic is formalized in the Isabelle proof assistant for the deductive verification of EVM bytecode. The program logic is proven sound based on the semantic foundation provided in [19]. On the down side, it is relatively difficult to specify the desired properties and auxiliary information needed for a proof, while working with low-level code. In consideration of this issue, [23] proposes to conduct theorem proving based verification of Ethereum smart contract at the level of the Ethereum intermediate language Yul.

Program Verification via Reachability Logics In [29], an approach and tool for verifying programs directly based on an operational semantics is proposed. The work builds on matching logic. It has a language-agnostic proof system using a unified representation of both the language semantics and the program correctness specifications. This approach has proven to be effective for real-world programming languages. If the approach is taken for the verification of smart contracts, there will be no need to develop a program logic and prove its correspondence with the semantics. On the other hand, the specification of the correctness assertions and the intermediate assertions (e.g., contract invariants and loop invariants) in matching logic patterns is expected to be more verbose on average than that in the proposed program logic. There will also be the need for an operational semantics in which the calls to contracts with unknown code is specified abstractly. It remains to be an interesting topic to evaluate the approach of [29] in the formal verification of smart contracts, and to figure out about the impact of the underlying logic (language-specific program logic VS language-agnostic matching logic) on the conceptual complexity in reasoning about smart contracts and on the efficiency of verifiers that can be implemented.

7 Conclusion

Atomicity guarantees are crucial for the integrity of smart contracts. We propose an atomicity criterion that supports the characterization of semantic updates to variables, and enables the flexible specification of the operations that are supposed to be performed in atomic batches. We devise a Hoare-style program logic that supports the sound verification of the proposed atomicity criterion on smart contracts specifically, and partial correctness properties of smart contracts in general. The program logic is devised for a core Solidity-like programming language supporting the use of local and state variables, mappings, intra-contract and inter-contract calls, ether transfers, and the handling of exceptions. We illustrate the advantages of the proposed atomicity criterion, and the effectiveness of the program logic, using examples with practical relevance.

In the contract invariants of the proposed program logic, the Boolean conditions that are preserved as well as which state variables are unaffected by

arbitrary executions of a smart contract can be expressed. However, the specification of the constraints on the value changes of state variables is not supported. In future work, we plan to further improve the expressiveness of the contract invariants, and extend our development to handle contract creation.

Acknowledgments. This work was supported by the National Natural Science Foundation of China (61877040, 61876111, 62002246), and the general project numbered KM202010028010 of Beijing Municipal Education Commission. We thank the anonymous reviewers for their invaluable feedback on this work and its presentation.

References

1. Ethereum smart contract best practices – known attacks. `https://consensys.github.io/smart-contract-best-practices/known_attacks/`.
2. Solidity. `https://solidity.readthedocs.io/en/v0.6.10/`.
3. S. Amani, M. Bgel, M. Bortin, and M. Staples. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *CPP'18*, pages 66–77.
4. K. R. Apt. Ten years of Hoare's logic: A survey - part 1. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, 1981.
5. N. Atzei, M. Bartoletti, and T. Cimoli. A survey of attacks on Ethereum smart contracts (SoK). In *POST'17*, pages 164–186.
6. M. Bartoletti, L. Galletta, and M. Murgia. A minimal core calculus for Solidity contracts. *CoRR*, abs/1710.09437, 2019.
7. B. Beckert, M. Herda, M. Kirsten, and J. Schiffl. Formal specification and verification of Hyperledger Fabric chaincode. In *SDLT'18*.
8. B. Bernardo, R. Cauderlier, B. Pesin, and J. Tesson. Albert, an intermediate smart-contract language for the Tezos blockchain. *CoRR*, abs/2001.02630, 2020.
9. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, and S. Z. Béguelin. Formal verification of smart contracts: Short paper. In *PLAS@CCS'16*, pages 91–96.
10. S. Blackshear, E. Cheng, D. L. Dill, V. Gao, and B. Maurer. Move: A language with programmable resources. `https://developers.libra.org/`, 2020.
11. J. Chang, B. Gao, H. Xiao, J. Sun, Y. Cai, and Z. Yang. scompile: Critical path identification and analysis for smart contracts. In *ICFEM'19*, pages 286–304.
12. T. Chen, Y. Zhang, Z. Li, X. Luo, T. Wang, R. Cao, X. Xiao, and X. Zhang. Tokenscope: Automatically detecting inconsistent behaviors of cryptocurrency tokens in Ethereum. In *CCS'19*, pages 1503–1520.
13. S. Crafa, M. D. Pirro, and E. Zucca. Is Solidity solid enough? In *FC'19*, pages 138–153.
14. M. Gordon and H. Collavizza. Forward with Hoare. In *Reflections on the Work of C. A. R. Hoare*, pages 101–121. 2010.
15. I. Grishchenko, M. Maffei, and C. Schneidewind. Foundations and tools for the static analysis of Ethereum smart contracts. In *CAV'18*, pages 51–78.
16. I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In *POST'18*, pages 243–269.
17. Á. Hajdu and D. Jovanovic. Solc-verify: A modular verifier for Solidity smart contracts. In *VSTTE'19*, pages 161–179.

18. E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. KEVM: A complete formal semantics of the Ethereum virtual machine. In *CSF'18*, pages 204–217.
19. Y. Hirai. Defining the Ethereum virtual machine for interactive theorem provers. In *FC'17*, pages 520–535.
20. B. Jiang, Y. Liu, and W. K. Chan. Contractfuzzer: fuzzing smart contracts for vulnerability detection. In *ASE'18*, pages 259–269.
21. J. Jiao, S. Lin, and J. Sun. A generalized formal semantic framework for smart contracts. In *FASE'20*, pages 75–96.
22. S. Kalra, S. Goel, M. Dhawan, and S. Sharma. ZEUS: analyzing safety of smart contracts. In *NDSS'18*.
23. X. Li, Z. Shi, Q. Zhang, G. Wang, Y. Guan, and N. Han. Towards verifying ethereum smart contracts at intermediate language level. In Y. A. Ameur and S. Qin, editors, *ICFEM*, pages 121–137, 2019.
24. L. Luu, D. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. In *CCS'16*, pages 254–269.
25. I. Nikolic, A. Kolluri, I. Sergey, P. Saxena, and A. Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *ACSAC'18*, pages 653–663.
26. A. Permenev, D. Dimitrov, P. Tsankov, D. Drachsler-Cohen, and M. Vechev. Verx: Safety verification of smart contracts. In *S&P'20*.
27. C. Schneidewind, I. Grishchenko, M. Scherer, and M. Maffei. eThor: Practical and provably sound static analysis of Ethereum smart contracts. *ArXiv*, abs/2005.06227, 2020.
28. I. Sergey, V. Nagaraj, J. Johannsen, A. Kumar, A. Trunov, and K. C. G. Hao. Safer smart contract programming with Scilla. In *OOPSLA'19*, pages 1–30.
29. A. Stefanescu, D. Park, S. Yuwen, Y. Li, and G. Rosu. Semantics-based program verifiers for all languages. In *OOPSLA'16*, pages 74–91.
30. N. Szabo. Smart contracts. `https://nakamotoinstitute.org/formalizing-securing-relationships/`, 1994.
31. P. Tolmach, Y. Li, S.-W. Lin, Y. Liu, and Z. Li. A survey of smart contract formal specification and verification. *CoRR*, arXiv:2008.02712, 2020.
32. G. Wood. Ethereum: A secure decentralised generlised transaction ledger. `https://gavwood.com/paper.pdf`.
33. D. Yaga, P. Mell, N. Roby, and K. Scarfone. Blockchain technology overview. Technical report, NISTIR 8202, 2018.
34. J. Zakrzewski. Towards verification of Ethereum smart contracts: A formalization of core of Solidity. In *VSTTE'18*, pages 229–247.

In the two appendices below, we provide further details of our technical development.

- In Appendix A, we provide the complete definition of the type system and the semantics for our smart contract language. We also sketch the proof of Theorem 1. This is relevant for Section 3 of the main text.
- In Appendix B, we provide the complete definition of $[e]^{\mathsf{c}}_X$, of the interpretation of assertions, of the update of assertion states, and of syntactical substitution. We provide the rules of the program logic missing from the main text. We also sketch the proof of Theorem 2. This is relevant for Section 4 of the main text.

A Supplementary Material for Section 3

A.1 Definition of the Type System

The typing judgments are as follows.

$$\begin{aligned} \mathit{ctrs}, \mathit{ste}, \mathit{lte} &\vdash e : \mathit{tp} \\ \mathit{ctrs}, \mathit{ste}, \mathit{lte} &\vdash S \\ \mathit{ctrs}, \mathit{ste} &\vdash \mathit{fd} \\ \mathit{ctrs}, \mathit{ste} &\vdash \mathit{fbd} \\ \mathit{ctrs} &\vdash \mathit{ctr} \\ &\vdash \mathit{ctrs} \end{aligned}$$

The judgment for expressions says that the expression e has type tp, under a list of contract definitions ctrs, a storage typing environment ste, and a local typing environment lte. The judgment for statements says that the statement S is well-typed under the same environments as for expressions. The judgment for function definitions says that the function definition fd is well-typed under the list ctrs of contracts and the storage typing environment ste. The judgment for fallback function definitions admits analogous explanation. The judgment for contracts says that the contract ctr is well-typed under the given list ctrs of contracts. The judgment for lists of contracts says that the list ctrs of contracts is well-typed.

Let $[\![\theta]\!]$ be the set of all values for the basic type θ. We define

$$\begin{aligned} [\![\mathsf{U160}]\!] &:= \mathbb{N}_{160} \\ [\![\mathsf{U256}]\!] &:= \mathbb{N}_{256} \\ [\![\mathsf{bool}]\!] &:= \{\mathit{tt}, \mathit{ff}\} \end{aligned}$$

The typing rules for expressions are as follows.

$$\frac{v \in [\![\theta]\!]}{\mathit{ctrs}, \mathit{ste}, \mathit{lte} \vdash (v : \theta) : \theta}$$

$$\frac{x \in \mathit{dom}(\mathit{lte})}{\mathit{ctrs}, \mathit{ste}, \mathit{lte} \vdash x : \mathit{lte}(x)} \qquad \frac{x \notin \mathit{dom}(\mathit{lte}) \quad x \in \mathit{dom}(\mathit{ste})}{\mathit{ctrs}, \mathit{ste}, \mathit{lte} \vdash x : \mathit{ste}(x)}$$

$$\frac{\mathit{ctrs}, \mathit{ste}, \mathit{lte} \vdash x : \theta_1 \to \theta_2 \quad \mathit{ctrs}, \mathit{ste}, \mathit{lte} \vdash e : \theta_1}{\mathit{ctrs}, \mathit{ste}, \mathit{lte} \vdash x\langle e \rangle : \theta_2}$$

$$\frac{ctrs, ste, lte \vdash e_1 : \mathsf{U256} \quad ctrs, ste, lte \vdash e_2 : \mathsf{U256}}{ctrs, ste, lte \vdash e_1\ aop\ e_2 : \mathsf{U256}}$$

$$\frac{ctrs, ste, lte \vdash e_1 : \mathsf{U256} \quad ctrs, ste, lte \vdash e_2 : \mathsf{U256}}{ctrs, ste, lte \vdash e_1\ cop\ e_2 : \mathsf{bool}}$$

$$\frac{ctrs, ste, lte \vdash e_1 : \mathsf{bool} \quad ctrs, ste, lte \vdash e_2 : \mathsf{bool}}{ctrs, ste, lte \vdash e_1\ bop\ e_2 : \mathsf{bool}}$$

$$\frac{}{ctrs, ste, lte \vdash \mathsf{this} : \mathsf{U160}} \qquad \frac{}{ctrs, ste, lte \vdash \mathsf{caller} : \mathsf{U160}}$$

$$\frac{}{ctrs, ste, lte \vdash \mathsf{callval} : \mathsf{U256}} \qquad \frac{}{ctrs, ste, lte \vdash \mathsf{balance} : \mathsf{U256}}$$

$$\frac{c \in cids(ctrs)}{ctrs, ste, lte \vdash \mathsf{adr}(c) : \mathsf{U160}}$$

Hence, arithmetic operations and comparative operations are allowed for 256-bit unsigned integers, but disallowed for 160-bit unsigned integers that are used as account addresses.

The typing rules for statements are as follows.

$$\frac{}{ctrs, ste, lte \vdash \mathsf{skip}}$$

$$\frac{(\exists x : e_1 = x \vee \exists x, e' : e_1 = x\langle e' \rangle) \quad ctrs, ste, lte \vdash e_1 : \theta \quad ctrs, ste, lte \vdash e_2 : \theta}{ctrs, ste, lte \vdash e_1 := e_2}$$

$$\frac{ctrs, ste, lte \vdash e : \mathsf{bool} \quad ctrs, ste, lte \vdash S}{ctrs, ste, lte \vdash \mathsf{if}\ (e)\ \{S\}} \qquad \frac{ctrs, ste, lte \vdash e : \mathsf{bool} \quad ctrs, ste, lte \vdash S}{ctrs, ste, lte \vdash \mathsf{while}\ (e)\ \{S\}}$$

$$\frac{ctrs, ste, lte \vdash S_1 \quad ctrs, ste, lte \vdash S_2}{ctrs, ste, lte \vdash S_1; S_2}$$

$$\frac{\begin{array}{c} ctrs = [\ldots, \mathsf{contract}\ c\ \{\ \mathsf{var}\ tpvs\ fd_1, \ldots, fd_l\ fbd\ \}, \ldots] \\ i \in \{1, \ldots, l\} \quad fd_i = \mathsf{fun}\ f(x_1 : \theta_1, \ldots, x_n : \theta_n; x'_1 : \theta'_1, \ldots, x'_m : \theta'_m)\ \mathsf{ret}\ x' : \theta'\ \{S\} \\ \forall i \in \{1, \ldots, n\} : ctrs, ste, lte \vdash e_i : \theta_i \qquad lte(x) = \theta' \end{array}}{ctrs, ste, lte \vdash \mathsf{adr}(c).f(e_1, \ldots, e_n) \to x}$$

$$\frac{\begin{array}{c} \neg(\exists c : e_{\mathrm{a}} = \mathsf{adr}(c)) \quad ctrs, ste, lte \vdash e_{\mathrm{a}} : \mathsf{U160} \\ \forall i \in \{1, \ldots, n\} : ctrs, ste, lte \vdash e_i : \theta_i \quad x \in dom(lte) \end{array}}{ctrs, ste, lte \vdash e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x}$$

$$\frac{ctrs, ste, lte \vdash e_{\mathrm{a}} : \mathsf{U160} \quad ctrs, ste, lte \vdash e : \mathsf{U256}}{ctrs, ste, lte \vdash e_{\mathrm{a}}.\mathsf{transfer}(e)}$$

$$\frac{}{ctrs, ste, lte \vdash \mathsf{throw}} \qquad \frac{ctrs, ste, lte \vdash S_1 \quad ctrs, ste, lte \vdash S_2}{ctrs, ste, lte \vdash \mathsf{try}\ \{S_1\}\ \mathsf{catch}\ \{S_2\}}$$

In the premises for the rules for function calls, it is required that x should be in the domain of lte. This essentially requires the return values from function calls to be written into local variables first. This simplifying condition imposes no real restriction, because the return data already written into a local variable may then go into the storage of the current contract, or be used in further computation.

The typing rule for function definitions is as follows.

$$\frac{ctrs, ste, [x_1 \mapsto \theta_1, \ldots, x_n \mapsto \theta_n, x'_1 \mapsto \theta'_1, \ldots, x'_m \mapsto \theta'_m, x \mapsto \theta] \vdash S}{ctrs, ste \vdash \mathsf{fun}\ f(x_1 : \theta_1, \ldots, x_n : \theta_n; x'_1 : \theta'_1, \ldots, x'_m : \theta'_m)\ \mathsf{ret}\ x : \theta\ \{S\}}$$

The typing rule for fallback function definitions is as follows.

$$\frac{ctrs, ste, [x_1 \mapsto \theta_1, \ldots, x_n \mapsto \theta_n] \vdash S}{ctrs, ste \vdash \mathsf{fun}\ (x_1 : \theta_1, \ldots, x_n : \theta_n)\ \{S\}}$$

The typing rule for contract definitions is as follows.

$$\frac{\begin{array}{c} ctr = \mathsf{contract}\ c\ \{\ \mathsf{var}\ x_1 : tp_1, \ldots, x_n : tp_n\ fd_1\ \ldots\ fd_m\ fbd\ \} \\ \forall i, j \in \{1, \ldots, n\} : x_i \neq x_j \qquad \forall i, j \in \{1, \ldots, m\} : fd_i.f \neq fd_j.f \\ ste = [x_1 \mapsto tp_1, \ldots, x_n \mapsto tp_n] \quad ctrs, ste \vdash fbd \quad \forall i \in \{1, \ldots, m\} : ctrs, ste \vdash fd_i \end{array}}{ctrs \vdash ctr}$$

The typing rule for lists of contracts is as follows.

$$\frac{\begin{array}{c} ctrs = [ctr_1, \ldots, ctr_n] \qquad \forall i \in \{1, \ldots, n\} :\ ctrs \vdash ctr_i \\ \forall i, j \in \{1, \ldots, n\} : i \neq j \Rightarrow cid(ctr_i) \neq cid(ctr_j) \end{array}}{\vdash ctrs}$$

A.2 Definition of the Semantics

Judgments The semantics of the smart contract language is given as calculus rules for the following judgments.

$$\langle e, lst, wst, ee \rangle \downarrow^{\eta} l$$

$$\langle S, lst, wst, ee \rangle \rightarrow_{\eta} (lst', wst', cs)_{flg}$$

The first judgment says, the expression e is evaluated to the literal l under the local state lst, the world state wst, the execution environment ee, and the address environment η. The second judgment says: the evaluation of the statement S in the local state lst, the world state wst, the execution environment ee, and the address environment η results in the local state lst', the world state wst', the set cs of successful calls, and the Boolean flag flg indicating the existence of uncaught exceptions.

Semantic Rules for Expressions We assume that for an arithmetic operator aop, $[\![aop]\!]$ is a function in $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$. We assume that for a comparative operator cop, $[\![cop]\!]$ is a function in $\mathbb{N} \times \mathbb{N} \rightarrow \{tt, ff\}$. We assume that for a Boolean operator bop, $[\![bop]\!]$ is a function in $\{tt, ff\} \times \{tt, ff\} \rightarrow \{tt, ff\}$.

The semantic rules for expressions are as follows.

$$\frac{}{\langle v : \theta, lst, wst, ee\rangle \downarrow^{\eta} v : \theta}$$

$$\frac{x \in dom(lst)}{\langle x, lst, wst, ee\rangle \downarrow^{\eta} lst(x)}$$

$$\frac{\begin{array}{c} x \notin dom(lst) \quad wst(ee.ths) = ast \quad ast.st(x) = v \\ ast.ctr = \mathsf{contract}\ c\ \{\ \mathsf{var}\ \ldots\, x : \theta \ldots\ fds\ fbd\ \} \end{array}}{\langle x, lst, wst, ee\rangle \downarrow^{\eta} v : \theta}$$

$$\frac{\begin{array}{c} x \notin dom(lst) \quad wst(ee.ths) = ast \quad ast.st(x) = g \\ ast.ctr = \mathsf{contract}\ c\ \{\ \mathsf{var}\ \ldots\, x : \theta_1 \to \theta_2 \ldots\ fds\ fbd\ \} \quad \langle e, lst, wst, ee\rangle \downarrow^{\eta} v : \theta_1 \end{array}}{\langle x\langle e\rangle, lst, wst, ee\rangle \downarrow^{\eta} g(v) : \theta_2}$$

$$\frac{\langle e_1, lst, wst, ee\rangle \downarrow^{\eta} v_1 : \mathsf{U256} \quad \langle e_2, lst, wst, ee\rangle \downarrow^{\eta} v_2 : \mathsf{U256} \quad [\![aop]\!](v_1, v_2) = v}{\langle e_1\ aop\ e_2, lst, wst, ee\rangle \downarrow^{\eta} v\%|\mathbb{N}_{256}| : \mathsf{U256}}$$

$$\frac{\langle e_1, lst, wst, ee\rangle \downarrow^{\eta} v_1 : \mathsf{U256} \quad \langle e_2, lst, wst, ee\rangle \downarrow^{\eta} v_2 : \mathsf{U256} \quad [\![cop]\!](v_1, v_2) = v}{\langle e_1\ cop\ e_2, lst, wst, ee\rangle \downarrow^{\eta} v : \mathsf{bool}}$$

$$\frac{\langle e_1, lst, wst, ee\rangle \downarrow^{\eta} v_1 : \mathsf{bool} \quad \langle e_2, lst, wst, ee\rangle \downarrow^{\eta} v_2 : \mathsf{bool} \quad [\![bop]\!](v_1, v_2) = v}{\langle e_1\ bop\ e_2, lst, wst, ee\rangle \downarrow^{\eta} v : \mathsf{bool}}$$

$$\frac{}{\langle \mathsf{this}, lst, wst, ee\rangle \downarrow^{\eta} ee.ths : \mathsf{U160}}$$

$$\frac{}{\langle \mathsf{caller}, lst, wst, ee\rangle \downarrow^{\eta} ee.clr : \mathsf{U160}}$$

$$\frac{}{\langle \mathsf{callval}, lst, wst, ee\rangle \downarrow^{\eta} ee.cvl : \mathsf{U256}}$$

$$\frac{wst(ee.ths) = ast}{\langle \mathsf{balance}, lst, wst, ee\rangle \downarrow^{\eta} ast.bal : \mathsf{U256}}$$

$$\frac{}{\langle \mathsf{adr}(c), lst, wst, ee\rangle \downarrow^{\eta} \eta(c) : \mathsf{U160}}$$

In the above, the premises for the two rules for the evaluation of a variable x are formulated such that the evaluation result may not carry a function type. This reflects that we do not permit the evaluation of mapping variables in isolation. A mapping variable x must be evaluated as part of some expression $x\langle e\rangle$. In the rule for $e_1\ aop\ e_2$, modular arithmetics is specified. Hence, the result of evaluating $e_1\ aop\ e_2$ is guaranteed to be in $\mathbb{N}_{256}$.

Semantic Rules for Statements The semantic rules for skip and assignments are as follows.

$$\frac{}{\langle \mathsf{skip}, lst, wst, ee\rangle \to_\eta (lst, wst, \emptyset)_{\mathit{ff}}}$$

$$\frac{x \in dom(lst) \quad \langle e, lst, wst, ee\rangle \downarrow^\eta l}{\langle x := e, lst, wst, ee\rangle \to_\eta (lst[x \mapsto l], wst, \emptyset)_{\mathit{ff}}}$$

$$\frac{\begin{array}{c} x \notin dom(lst) \quad \langle e, lst, wst, ee\rangle \downarrow^\eta v : \theta \\ ast = wst(ee.ths) \quad wst' = wst[ee.ths \mapsto ast[st := ast.st[x \mapsto v]]] \end{array}}{\langle x := e, lst, wst, ee\rangle \to_\eta (lst, wst', \emptyset)_{\mathit{ff}}}$$

$$\frac{\begin{array}{c} x \notin dom(lst) \quad \langle [e_1, e_2], lst, wst, ee\rangle \downarrow^\eta [v_1 : \theta_1, v_2 : \theta_2] \\ ast = wst(ee.ths) \quad g = ast.st(x) \quad g' = g[v_1 \mapsto v_2] \\ wst' = wst[ee.ths \mapsto ast[st := ast.st[x \mapsto g']]] \end{array}}{\langle x\langle e_1\rangle := e_2, lst, wst, ee\rangle \to_\eta (lst, wst', \emptyset)_{\mathit{ff}}}$$

The semantic rules for the if statement and the while statement are as follows.

$$\frac{\langle e, lst, wst, ee\rangle \downarrow^\eta \mathit{tt} : \mathsf{bool} \quad \langle S, lst, wst, ee\rangle \to_\eta \gamma}{\langle \mathsf{if}\ (e)\ \{S\}, lst, wst, ee\rangle \to_\eta \gamma}$$

$$\frac{\langle e, lst, wst, ee\rangle \downarrow^\eta \mathit{ff} : \mathsf{bool}}{\langle \mathsf{if}\ (e)\ \{S\}, lst, wst, ee\rangle \to_\eta (lst, wst, \emptyset)_{\mathit{ff}}}$$

$$\frac{\begin{array}{c} \langle e, lst, wst, ee\rangle \downarrow^\eta \mathit{tt} : \mathsf{bool} \quad \langle S, lst, wst, ee\rangle \to_\eta (lst'', wst'', cs_1)_{\mathit{ff}} \\ \langle \mathsf{while}\ (e)\ \{S\}, lst'', wst'', ee\rangle \to_\eta (lst', wst', cs_2)_{\mathit{ftg}} \end{array}}{\langle \mathsf{while}\ (e)\ \{S\}, lst, wst, ee\rangle \to_\eta (lst', wst', cs_1 \cup cs_2)_{\mathit{ftg}}}$$

$$\frac{\langle e, lst, wst, ee\rangle \downarrow^\eta \mathit{tt} : \mathsf{bool} \quad \langle S, lst, wst, ee\rangle \to_\eta (lst', wst', cs)_{\mathit{tt}}}{\langle \mathsf{while}\ (e)\ \{S\}, lst, wst, ee\rangle \to_\eta (lst', wst', cs)_{\mathit{tt}}}$$

$$\frac{\langle e, lst, wst, ee\rangle \downarrow^\eta \mathit{ff} : \mathsf{bool}}{\langle \mathsf{while}\ (e)\ \{S\}, lst, wst, ee\rangle \to_\eta (lst, wst, \emptyset)_{\mathit{ff}}}$$

The rules for the while loop say that if a round of the loop finishes with an exception, then there will be no further effects of the states, and the loop finishes with an exception. If a round of the loop finishes normally, then the loop is continued. If the conditional expression is evaluated to false, then the loop terminates normally.

The rule for the statement throw is as follows.

$$\frac{}{\langle \mathsf{throw}, lst, wst, ee\rangle \to_\eta (lst, wst, \emptyset)_{\mathit{tt}}}$$

This rule says that the execution of throw sets the flag indicating an exception, without introducing other changes to the state.

Next, the rules for the statement $\mathsf{try}\ \{S_1\}\ \mathsf{catch}\ \{S_2\}$ and the sequential composition are collected together for a side-by-side comparison.

$$\frac{\langle S_1, lst, wst, ee\rangle \to_\eta (lst', wst', cs)_{\mathit{ff}}}{\langle \mathsf{try}\ \{S_1\}\ \mathsf{catch}\ \{S_2\}, lst, wst, ee\rangle \to_\eta (lst', wst', cs)_{\mathit{ff}}}$$

$$\frac{\langle S_1, lst, wst, ee\rangle \to_\eta (lst'', wst'', cs_1)_{tt} \quad \langle S_2, lst'', wst'', ee\rangle \to_\eta (lst', wst', cs_2)_{\mathit{flg}}}{\langle \mathsf{try}\ \{S_1\}\ \mathsf{catch}\ \{S_2\}, lst, wst, ee\rangle \to_\eta (lst', wst', cs_1 \cup cs_2)_{\mathit{flg}}}$$

$$\frac{\langle S_1, lst, wst, ee\rangle \to_\eta (lst'', wst'', cs_1)_{\mathit{ff}} \quad \langle S_2, lst'', wst'', ee\rangle \to_\eta (lst', wst', cs_2)_{\mathit{flg}}}{\langle S_1; S_2, lst, wst, ee\rangle \to_\eta (lst', wst', cs_1 \cup cs_2)_{\mathit{flg}}}$$

$$\frac{\langle S_1, lst, wst, ee\rangle \to_\eta (lst', wst', cs)_{tt}}{\langle S_1; S_2, lst, wst, ee\rangle \to_\eta (lst', wst', cs)_{tt}}$$

For a try construct, if the first statement finishes with an exception, then the second statement is evaluated in the state when the exception happens. For a sequential composition, if the first statement finishes normally, then the second statement is evaluated.

The following rule describes the evaluation of a function call such that the function with the intended signature exists in the target contract. In this case, the intended callee function is executed. The expression $d(\theta)$ is defined as the default value for the basic type θ. More concretely, $d(\mathsf{U160}){:=}0$, $d(\mathsf{U256}){:=}0$, and $d(\mathsf{bool}){:=}\mathit{ff}$.

$$\frac{\begin{array}{c}\langle [e_{\mathrm{a}}, e_1, \ldots, e_n], lst, wst, ee\rangle \downarrow^\eta [l, l_1, \ldots, l_n] \quad l = v : \mathsf{U160} \\ wst(v) = ast \in ASt \quad ast.ctr = \mathsf{contract}\ c\ \{\ \mathsf{var}\ tpvs\ \ldots fd \ldots fbd\ \} \\ fd = \mathsf{fun}\ f(x_1 : \theta_1, \ldots, x_n : \theta_n; x'_1 : \theta'_1, \ldots, x'_m : \theta'_m)\ \mathsf{ret}\ x' : \theta'\ \{S\} \\ \forall i \in \{1, \ldots, n\} : \exists v_i : l_i = v_i : \theta_i \qquad \exists v' : lst(x) = v' : \theta' \\ lst' = [x_1 \mapsto l_1, \ldots, x_n \mapsto l_n, x'_1 \mapsto d(\theta'_1) : \theta'_1, \ldots, x'_m \mapsto d(\theta'_m) : \theta'_m, x' \mapsto d(\theta') : \theta'] \\ ee' = (v, ee.ths, 0) \qquad \langle S, lst', wst, ee'\rangle \to_\eta (lst'', wst'', cs)_{\mathit{flg}} \\ \gamma \in \left(\begin{array}{c} \mathit{if}\ \neg \mathit{flg}\ \mathit{then}\ \{(lst[x \mapsto lst''(x')], wst'', cs \cup \{call(ee.ths, v, f)\})_{\mathit{ff}}, (lst, wst, \emptyset)_{tt}\} \\ \mathit{else}\ \ \{(lst, wst, \emptyset)_{tt}\} \end{array} \right) \end{array}}{\langle e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x, lst, wst, ee\rangle \to_\eta \gamma}$$

We explain how the semantics capture exceptions triggered by gas exhaustion in an abstract manner. In Ethereum, the execution of each instruction in the low-level code of a contract consumes gas. It is non-trivial to ascribe the amount of gas consumed to the execution of programming constructs of a high-level language. Hence, we leave out the concrete specification of the amount of gas consumed in the semantic rules. However, in function calls, we specify that even if the callee finishes without any exception (i.e., with $(lst', wst', cs)_{\mathit{ff}}$ as the execution result of the callee), the overall call might still end up with an exception. This captures gas exhaustion caused either by the call itself, or in the execution of the callee. For this reason, our semantics is non-deterministic, although the computation that can be performed by a smart contract is deterministic.

The following rule describes the evaluation of a function call such that no function with the intended signature exists in the target contract. In this case, the fallback function of the target contract is executed.

$$\frac{\begin{array}{c}\langle [e_{\mathrm{a}}, e_1, \ldots, e_n], lst, wst, ee\rangle \downarrow^{\eta} [l, l_1, \ldots, l_n] \quad l = v : \mathsf{U160} \\ wst(v) = ast \in ASt \quad ast.ctr = \mathsf{contract}\ c\ \{\ \mathsf{var}\ tvs\ fd_1 \ldots fd_m\ fbd\ \} \\ \neg(\exists i, \theta_1, \ldots, \theta_n, \theta', x_i, \ldots, x_n, x' : fd_i = \mathsf{fun}\ f(x_1 : \theta_1, \ldots, x_n : \theta_n; \ldots)\ \mathsf{ret}\ x' : \theta'\ \{\ldots\} \\ \qquad \wedge\ \forall j : \exists v_j : l_j = v_j : \theta_j\ \wedge\ \exists v' : lst(x) = v' : \theta') \\ fbd = \mathsf{fun}\ (x_1 : \theta_1, \ldots, x_n : \theta_n)\ \{S\} \qquad lst' = [x_1 \mapsto d(\theta_1) : \theta_1, \ldots, x_n \mapsto d(\theta_n) : \theta_n] \\ ee' = (v, ee.ths, 0) \qquad \langle S, lst', wst, ee'\rangle \rightarrow_{\eta} (lst'', wst'', cs)_{flg} \\ \gamma \in \begin{pmatrix} if\ \neg flg\ then\ \{(lst, wst'', cs \cup \{call(ee.ths, v, f)\})_{ff}, (lst, wst, \emptyset)_{tt}\} \\ else\ \ \{(lst, wst, \emptyset)_{tt}\} \end{pmatrix}\end{array}}{\langle e_{\mathrm{a}}.f(e_1, \ldots, e_n) \rightarrow x, lst, wst, ee\rangle \rightarrow_{\eta} \gamma}$$

The following rule describes the evaluation of a function call such that no account exists at the target address. The call results in an exception.

$$\frac{\langle e_{\mathrm{a}}, lst, wst, ee\rangle \downarrow^{\eta} v : \mathsf{U160} \quad wst(v) = \bot}{\langle e_{\mathrm{a}}.f(e_1, \ldots, e_n) \rightarrow x, lst, wst, ee\rangle \rightarrow_{\eta} (lst, wst, \emptyset)_{tt}}$$

The rule above describes the evaluation of a transfer such that an account exists at the target address (which is different from the address of the current account), the transfered amount does not exceed the balance of the current contract, and does not cause an overflow at the receiver's side. The balances of the current account and the target account are updated after the transfer. The fallback function at the target contract is executed.

$$\frac{\begin{array}{c}\langle [e_{\mathrm{a}}, e], lst, wst, ee\rangle \downarrow^{\eta} [l_{\mathrm{a}}, l] \quad l_{\mathrm{a}} = v : \mathsf{U160} \quad v \neq ee.ths \quad wst(v) = ast \in ASt \\ ast.ctr = \mathsf{contract}\ c\ \{\ \mathsf{var}\ tvs\ fds\ fbd\ \} \quad fbd = \mathsf{fun}\ (x_1 : \theta_1, \ldots, x_n : \theta_n)\ \{S\} \\ l = v' : \mathsf{U256} \qquad wst(ee.ths) = ast_0 \in ASt \\ v' \leq ast_0.bal \quad v' + ast.bal < |\mathbb{N}_{256}| \quad ee' = (v, ee.ths, v') \\ wst' = wst[ee.ths \mapsto ast_0[bal := ast_0.bal - v']][v \mapsto ast[bal := ast.bal + v']] \\ lst' = [x_1 \mapsto d(\theta_1) : \theta_1, \ldots, x_n \mapsto d(\theta_n) : \theta_n] \qquad \langle S, lst', wst', ee'\rangle \rightarrow_{\eta} (lst'', wst'', cs)_{flg} \\ \gamma \in \begin{pmatrix} if\ \neg flg\ then\ \{(lst, wst'', cs \cup \{call(ee.ths, v, \bot)\})_{ff}, (lst, wst, \emptyset)_{tt}\} \\ else\ \ \{(lst, wst, \emptyset)_{tt}\} \end{pmatrix}\end{array}}{\langle e_{\mathrm{a}}.\mathsf{transfer}(e), lst, wst, ee\rangle \rightarrow_{\eta} \gamma}$$

The following rule describes the evaluation of a transfer such that an account exists at the target address, but the transfered amount does not satisfy the constraints with respect to the balances at the source and destination accounts. An exception is resulted from this evaluation.

$$\frac{\begin{array}{c}\langle [e_{\mathrm{a}}, e], lst, wst, ee\rangle \downarrow^{\eta} [l_{\mathrm{a}}, l] \quad l_{\mathrm{a}} = v : \mathsf{U160} \quad v \neq ee.ths \quad wst(v) = ast \in ASt \\ l = v' : \mathsf{U256} \quad wst(ee.ths) = ast_0 \in ASt \quad v' > ast_0.bal \vee v' + ast.bal \geq |\mathbb{N}_{256}|\end{array}}{\langle e_{\mathrm{a}}.\mathsf{transfer}(e), lst, wst, ee\rangle \rightarrow_{\eta} (lst, wst, \emptyset)_{tt}}$$

The following rule describes the evaluation of a transfer such that no account exists at the target address. The evaluation results in an exception.

$$\frac{\langle e_{\mathrm{a}}, lst, wst, ee\rangle \downarrow^{\eta} v : \mathsf{U160} \quad v = ee.ths \vee wst(v) = \bot}{\langle e_{\mathrm{a}}.\mathsf{transfer}(e), lst, wst, ee\rangle \rightarrow_{\eta} (lst, wst, \emptyset)_{tt}}$$

A.3 Preservation of Types by Evaluation

We first provide a few definitions that are missing from the main text.

We define the function *ste-of* that forms a storage type environment based on the declaration of state variables in a contract.

Definition 1. *The function* $\mathit{ste\text{-}of} \in \mathit{Ctr} \to \mathit{STe}$ *is given by*

$$\mathit{ste\text{-}of}(\mathit{ctr}) := [x_1 \mapsto \mathit{tp}_1, \ldots, x_n \mapsto \mathit{tp}_n]$$

where $x_1, \ldots, x_n$, *and* $\mathit{tp}_1, \ldots, \mathit{tp}_n$ *satisfy*

$$\exists c, \mathit{fds}, \mathit{fbd} : \mathit{ctr} = \mathsf{contract}\ c\ \{\, \mathsf{var}\ x_1 : \mathit{tp}_1, \ldots, x_n : \mathit{tp}_n\ \mathit{fds}\ \mathit{fbd}\,\}$$

The function *lte-of* forms a local typing environment based on the declaration in a function, or a fallback function, in a well-typed contract.

Definition 2. *The function* $\mathit{lte\text{-}of} \in \mathit{Ctr} \times (\mathit{Fid} \cup \{\bot\}) \to \mathit{LTe} \cup \{\bot\}$ *is given by*

$$\mathit{lte\text{-}of}(\mathit{ctr}, f_\bot) := \begin{cases} [\vec{x} \mapsto \vec{\theta}, \vec{x'} \mapsto \vec{\theta'}, x_\mathrm{r} \mapsto \theta_\mathrm{r}] & \text{if } \exists f \in \mathit{Fid} : f_\bot = f \wedge \\ & \quad \exists c, \mathit{tpvs}, \mathit{fds}, \mathit{fbd} : \mathit{ctr} = \mathsf{contract}\ c\,\{\,\mathsf{var}\ \mathit{tpvs}\ \mathit{fds}\ \mathit{fbd}\,\} \wedge \\ & \quad \exists i, S : \mathit{fds}!i = \mathsf{fun}\ f(\overrightarrow{x:\theta}; \overrightarrow{x':\theta'})\ \mathsf{ret}\ x_\mathrm{r} : \theta_\mathrm{r}\ \{S\} \\ [\vec{x} \mapsto \vec{\theta}] & \text{if } f_\bot = \bot \wedge \\ & \quad \exists c, \mathit{tpvs}, \mathit{fds}, \mathit{fbd} : \mathit{ctr} = \mathsf{contract}\ c\,\{\,\mathsf{var}\ \mathit{tpvs}\ \mathit{fds}\ \mathit{fbd}\,\} \wedge \\ & \quad \exists S : \mathit{fbd} = \mathsf{fun}\ (\overrightarrow{x:\theta})\ \mathsf{ret}\ \ \{S\} \\ \bot & \text{otherwise} \end{cases}$$

Here, we write $f_\bot$ to represent an optional function identifier. For well-typed contracts, the function identifiers are required to be distinct. Hence, the function *lte-of* is well-defined.

We define the relation $\blacktriangleright$ to express the consistency of storage typing environments and storages.

Definition 3. *The binary relation* $\blacktriangleright$ *is given by*

$$\begin{aligned} \mathit{ste} \blacktriangleright \mathit{st} := \forall x \in \mathit{dom}(\mathit{ste}) : & (\exists \theta : \mathit{ste}(x) = \theta \wedge \mathit{st}(x) \in [\![\theta]\!]) \vee \\ & (\exists \theta_1, \theta_2 : \mathit{ste}(x) = \theta_1 \to \theta_2 \wedge \mathit{st}(x) \in [\![\theta_1]\!] \to [\![\theta_2]\!]) \end{aligned}$$

We define the relation $\triangleright$ to express the consistency of local typing environments and local states.

Definition 4. *The binary relation* $\triangleright$ *is given by*

$$\begin{aligned} \mathit{lte} \triangleright \mathit{lst} := & \mathit{dom}(\mathit{lte}) = \mathit{dom}(\mathit{lst}) \wedge \\ & (\forall x \in \mathit{dom}(\mathit{lst}) : \forall \theta : \mathit{lte}(x) = \theta \Rightarrow \exists v : \mathit{lst}(x) = v : \theta \wedge v \in [\![\theta]\!]) \end{aligned}$$

We proceed with establishing Theorem 1. In the following, we write $\mathit{wf}_{\mathrm{wst}}(\mathit{wst})$ to express

$$\forall v \in \mathbb{N}_{160} : \forall \mathit{ast} : \mathit{wst}(v) = \mathit{ast} \Rightarrow \mathit{ste\text{-}of}(\mathit{ast.ctr}) \blacktriangleright \mathit{ast.st} \wedge \exists \mathit{ctrs} : \mathit{ctrs} \vdash \mathit{ast.ctr}$$

Hence, $wf_{\text{wst}}(wst)$ imposes part of the requirement that is imposed in $wf(ctr, f_\perp, lst, wst)$, and $wf_{\text{wst}}(wst)$ is implied by $wf(ctr, f_\perp, lst, wst)$.

The following lemma says that the type of an expression is preserved by its evaluation result.

Lemma 1. *If $wf_{\text{wst}}(wst)$ holds, $ctrs, ste, lte \vdash e : tp$ can be derived, $ste = ste\text{-}of(wst(ee.ths).ctr)$, $lte \triangleright lst$, and $\langle e, lst, wst, ee\rangle \downarrow^{\eta} l$ can be derived, then $ctrs, ste, lte \vdash l : tp$ can be derived for any ste and lte.*

Proof (Sketch). The proof goes by induction on the derivation of the semantic evaluation.

We present the representative case where a variable $x \notin dom(lst)$ is evaluated, we have $x \notin dom(lte)$ because of $lte \triangleright lst$. Hence, we have $tp = ste(x)$ from the typing of x. Hence, we have

$$tp = ste\text{-}of(wst(ee.ths).ctr)(x) \tag{1}$$

according to the hypotheses of the lemma. Hence, x is declared as $x : tp$ in the contract $wst(ee.ths).ctr$, by the definition of *ste-of*. Since x is evaluated to some literal, and $x \notin dom(lst)$, we have $\langle x, lst, wst, ee\rangle \downarrow^{\eta} v : tp$ where

$$v = wst(ee.ths).st(x) \tag{2}$$

and tp is a basic type, according to the semantic rule for the evaluation of state variables. Specializing $wf_{\text{wst}}(wst)$ with $ee.ths$ and using (1) and (2), it can be deduced that $v \in [\![tp]\!]$. Hence, the literal $v : tp$ is well-typed with the type tp.

The case for the evaluation of $x\langle e\rangle$ is analogous to the case above. The case for the evaluation of a variable $x \in dom(lst)$, as well as the other cases, are straightforward.

Lemma 2. *If $wf_{\text{wst}}(wst)$ holds, $ctrs, ste, lte \vdash S$ can be derived, $lte \triangleright lst$, $ste = ste\text{-}of(wst(ee.ths).ctr)$, and $\langle S, lst, wst, ee\rangle \rightarrow_{\eta} (lst', wst', cs)_{flg}$ can be derived, then we have $wf_{\text{wst}}(wst')$ and $lte \triangleright lst'$.*

Proof (Sketch). The proof goes by induction on the derivation of the semantic evaluation for statements. We present a number of representative cases.

Case $\langle x := e, lst, wst, ee\rangle \rightarrow_{\eta} (lst, wst', \emptyset)_{ff}$, where $x \notin dom(lst)$. We have $lte \triangleright lst'$ because $lst' = lst$. We next show that $wf_{\text{wst}}(wst')$ holds. We have

$$wst' = wst[ee.ths \mapsto ast[st := ast.st[x \mapsto v]]] \tag{3}$$

$$\langle e, lst, wst, ee\rangle \downarrow^{\eta} v : \theta \tag{4}$$

according to the semantics, where $ast = wst(ee.ths)$, and $\theta \in \Theta$. We have $ctrs, ste, lte \vdash x : \theta'$, and

$$ctrs, ste, lte \vdash e : \theta' \tag{5}$$

for some θ', by the typing of $x := e$. Hence, using Lemma 1 and (4), we deduce that $v : \theta$ is well-typed, with type θ'. Hence, we have $\theta' = \theta$, and

$v \in [\![\theta]\!]$, using the typing rules for expressions. From the typing rules for expressions, $x \notin dom(lst)$, $lte \triangleright lst$, (5), and $\theta = \theta'$, we have $ste(x) = \theta$. Hence, we have $ste\text{-}of(wst(ee.ths).ctr)(x) = \theta$ using the hypotheses of the lemma. We have $wst(ee.ths).st(x) = v$ using (3). Hence, using $v \in [\![\theta]\!]$, and $wf_{\text{wst}}(wst)$, $wf_{\text{wst}}(wst')$ can be established, because no other update than $[x \mapsto v]$ happens.

Case $\langle \mathsf{while}\ (e)\ \{S_1\}, lst, wst, ee\rangle \rightarrow_\eta (lst', wst', cs_1 \cup cs_2)_{\mathit{fl}g}$, with e evaluated to true,

$$\langle S, lst, wst, ee\rangle \rightarrow_\eta (lst'', wst'', cs_1)_{\mathit{ff}} \text{ and} \tag{6}$$

$$\langle \mathsf{while}\ (e)\ \{S_1\}, lst'', wst'', ee\rangle \rightarrow_\eta (lst', wst', cs_2)_{\mathit{fl}g} \tag{7}$$

From the typing of the while statement, we have $ctrs, ste, lte \vdash S_1$. Using the IH on (6), we deduce $wf_{\text{wst}}(wst'')$ and $lte \triangleright lst''$. Using the IH again on (7), we have $wf_{\text{wst}}(wst')$ and $lte \triangleright lst'$.

Case $\langle e_{\text{a}}.f(e_1, \ldots, e_n) \rightarrow x, lst, wst, ee\rangle \rightarrow_\eta \gamma$, with the callee function existing in the target contract. If $\gamma = (lst, wst, \emptyset)_{tt}$, then it is trivial to obtain the desired conclusion. We next consider the case where

$$\gamma = (lst[x \mapsto lst''(x')], wst'', cs \cup \{call(ee, v, f)\})_{\mathit{ff}}$$

for some lst'', wst'' and cs such that $(lst'', wst'', cs)_{\mathit{ff}}$ is the evaluation result of the body of the callee, and some x' that is the return variable of the callee. Using $wf_{\text{wst}}(wst)$, and that there exists an account at the target address v (from the evaluation result of e_{a}), the target contract is typable. Hence, the type system ensures that there exists some $ctrs'$, such that

$$ctrs', ste', lte' \vdash S_1 \tag{8}$$

where S_1 is the function body of the callee,

$$ste' = ste\text{-}of(wst(v).ctr), \tag{9}$$

and $lte' = [x_1 \mapsto \theta_1, \ldots, x_n \mapsto \theta_n, x'_1 \mapsto \theta'_1, \ldots, x'_n \mapsto \theta'_m, x' \mapsto \theta']$, where x_1, $\ldots$, x_n are the formal parameters of the callee function, declared with the types θ_1, $\ldots$, θ_n, x'_1, $\ldots$, x'_m are the local variables of the callee function, declared with the types θ'_1, $\ldots$, θ'_m, and x' is the return variable of the callee that is declared with the type θ'. According to the corresponding semantic rule for the call, we have $v = ee'.ths$ where ee' is the execution environment in which the evaluation of S_1 takes place. Hence, we have

$$ste' = ste\text{-}of(wst(ee'.ths).ctr) \tag{10}$$

using (9). We have $\langle e_j, lst, wst, ee\rangle \downarrow^\eta v_j : \theta_j$ for each $j \in \{1, \ldots, n\}$ according to the semantics. We have $ctrs, ste, lte \vdash e_j : \theta'_j$ for some θ'_j from the typing of the overall call $e_{\text{a}}.f(e_1, \ldots, e_n) \rightarrow x$. Hence, we have $\theta'_j = \theta_j$, and $v_j : \theta_j$ is well-typed with the type θ_j, using Lemma 1 and the hypotheses of the current lemma. Hence, we have $v_j \in [\![\theta_j]\!]$. It can now be established that $lte' \triangleright lst'$

where lst' is the local state in which S_1 is evaluated. Hence, using the IH on $\langle S_1, lst', wst, ee' \rangle \rightarrow_\eta (lst'', wst'', cs)_{\not\!\!fl g}$, with (8) and (10), we have $wf_{\text{wst}}(wst'')$ and $lte' \triangleright lst''$. Since wst'' is the world state in the final evaluation result of the overall call, it only remains to establish $lte \triangleright lst[x \mapsto lst''(x')]$. According to the semantic rule for the call (relevant for this case), we have $lst(x) = v' : \theta'$ for some v'. Hence, we have $lte(x) = \theta'$ because of $lte \triangleright lst$. We have $lst''(x') = v'' : \theta'$ for some v'' such that $v'' \in [\![\theta']\!]$ according to $lte' \triangleright lst''$ and the definition of lte'. Hence, $lte \triangleright lst[x \mapsto lst''(x')]$ can be established using $lte \triangleright lst$.

The reasoning for the remaining cases follows analogous patterns as in one of the cases presented above.

Proof of Theorem 1 Theorem 1 of Section 3 can now be proven using Lemma 2. We assume $wf(ctr, f_\perp, lst, wst)$ from the hypotheses of Theorem 1. We then have

$$wf_{\text{wst}}(wst) \tag{11}$$

$$lte\text{-}of(ctr, f_\perp) \triangleright lst \tag{12}$$

Let lte_0:=$lte\text{-}of(ctr, f_\perp)$. Let ste_0:=$ste\text{-}of(wst(ee.ths).ctr)$. We then have $ste_0 = ste\text{-}of(ctr)$ using the hypotheses of this theorem. We have that $wst(ee.ths).ctr$ is a smart contract (i.e., not $\perp$) again using the hypotheses of this theorem. Hence, we have $wst(ee.ths) \neq \perp$. Hence, we obtain $ctrs' \vdash ctr$ for some $ctrs'$ using $wf_{\text{wst}}(wst)$ and the fact that $ctr = wst(ee.ths).ctr$. Hence, we have $ctrs', ste_0, lte_0 \vdash S$ because of $S = stmt\text{-}of(ctr, f_\perp)$. Using Lemma 2, (11), (12), and the definitions of lte_0 and ste_0, we obtain $wf_{\text{wst}}(wst')$ and $lte_0 \triangleright lst'$. It can thus be deduced that $wf(ctr, f_\perp, lst', wst')$. □

B Supplementary Material for Section 4

B.1 Definitions Related to Assertions

We define $[e]^{\mathrm{c}}_X$ as follows.

$$
\begin{aligned}
[v:\theta]^{\mathrm{c}}_X &:= v \\
[x]^{\mathrm{c}}_X &:= \begin{cases} x & \text{if } x \in X \\ x^c & \text{if } x \notin X \end{cases} \\
[x\langle e\rangle]^{\mathrm{c}}_X &:= x^c([e]^{\mathrm{c}}_X) \\
[e_1\ aop\ e_2]^{\mathrm{c}}_X &:= [e_1]^{\mathrm{c}}_X\ aop\ [e_2]^{\mathrm{c}}_X \\
[e_1\ cop\ e_2]^{\mathrm{c}}_X &:= [e_1]^{\mathrm{c}}_X\ cop\ [e_2]^{\mathrm{c}}_X \\
[e_1\ bop\ e_2]^{\mathrm{c}}_X &:= [e_1]^{\mathrm{c}}_X\ bop\ [e_2]^{\mathrm{c}}_X \\
[\mathsf{this}]^{\mathrm{c}}_X &:= \mathit{ths} \\
[\mathsf{caller}]^{\mathrm{c}}_X &:= \mathit{clr} \\
[\mathsf{callval}]^{\mathrm{c}}_X &:= \mathit{cvl} \\
[\mathsf{balance}]^{\mathrm{c}}_X &:= \mathfrak{b}(\overline{c}) \\
[\mathsf{adr}(c')]^{\mathrm{c}}_X &:= \overline{c'}
\end{aligned}
$$

We define the update of assertion states as follows. The clauses give a function in $\Sigma \to \mathit{BTrm} \to V \to (\Sigma \cup \{\bot\})$. Here, $V{:=}\mathbb{N}_{256} \cup \{\mathit{tt}, \mathit{ff}\} \cup (\mathbb{N}_{160} \to \mathbb{N}_{256}) \cup (\mathbb{N}_{160} \times \mathbb{N}_{160} \times (\mathit{Fid} \cup \{\bot\}))$, and V is ranged over by ν. In case there is no defining clause covering certain argument values, the result of the function is $\bot$ on those argument values. For the second argument, all basic terms but values are covered by the clauses.

$$
\begin{aligned}
\sigma[w \mapsto \nu] &:= \sigma[\zeta := \sigma.\zeta[w \mapsto \nu]] \\
\sigma[u \mapsto \nu] &:= \sigma[\zeta := \sigma.\zeta[u \mapsto \nu]] \\
\sigma[x \mapsto v] &:= \begin{cases} \sigma[\mathit{lst} := \sigma.\mathit{lst}[x \mapsto v:\theta]] & \text{if } \exists v' : \sigma.\mathit{lst}(x) = v':\theta \\ \bot & \text{otherwise} \end{cases} \\
\sigma[x^c \mapsto v] &:= \begin{cases} \sigma[\mathit{wst} := \sigma.\mathit{wst}[\eta(c) \mapsto \mathit{ast}[\mathit{st} := \mathit{ast}.\mathit{st}[x \mapsto v]]]] & \text{if } \sigma.\mathit{wst}(\eta(c)) = \mathit{ast} \\ \bot & \text{if } \sigma.\mathit{wst}(\eta(c)) = \bot \end{cases} \\
\sigma[\mathfrak{b} \mapsto \nu] &:= \sigma\left[\mathit{wst} := \lambda v. \begin{cases} \mathit{ast}[\mathit{bal} := \nu(v)] & \text{if } \sigma.\mathit{wst}(v) = \mathit{ast} \\ \bot & \text{if } \sigma.\mathit{wst}(v) = \bot \end{cases}\right] \\
&\quad (\text{where } \nu \in \mathbb{N}_{160} \to \mathbb{N}_{256}) \\
\sigma[\kappa \mapsto \nu] &:= \sigma[\mathit{cs} := \{\mathit{call}(v_1, v_2, f_\bot) \mid \nu(v_1, v_2, f_\bot) = \mathit{tt} \wedge v_1, v_2 \in \mathbb{N}_{160}\}] \\
&\quad (\text{where } \nu \in (\mathbb{N}_{160} \times \mathbb{N}_{160} \times (\mathit{Fid} \cup \{\bot\})) \to \{\mathit{tt}, \mathit{ff}\}) \\
\sigma[\mathit{ths} \mapsto v] &:= \sigma[\mathit{ee} := \sigma.\mathit{ee}[\mathit{ths} := v]] \\
\sigma[\mathit{clr} \mapsto v] &:= \sigma[\mathit{ee} := \sigma.\mathit{ee}[\mathit{clr} := v]] \\
\sigma[\mathit{cvl} \mapsto v] &:= \sigma[\mathit{ee} := \sigma.\mathit{ee}[\mathit{cvl} := v]] \\
\sigma[\varepsilon \mapsto v] &:= \sigma[\mathit{flg} := v]
\end{aligned}
$$

We next define the interpretation of assertions.

We define the interpretation of basic terms as follows.

$$
\begin{aligned}
[\![\nu]\!]_{\mathrm{btrm}}(\sigma) &:= \nu \\
[\![w]\!]_{\mathrm{btrm}}(\sigma) &:= (\sigma.\zeta)(w) \\
[\![x]\!]_{\mathrm{btrm}}(\sigma) &:= \begin{cases} v & \text{if } (\sigma.lst)(x) = v : \theta \\ \bot & \text{otherwise} \end{cases} \\
[\![x^c]\!]_{\mathrm{btrm}}(\sigma) &:= (\sigma.wst(\eta(c)).st)(x) \\
[\![\mathfrak{b}]\!]_{\mathrm{btrm}}(\sigma) &:= \lambda v. \begin{cases} ast.bal & \text{if } (\sigma.wst)(v) = ast \wedge ast \in ASt \\ \bot & \text{if } (\sigma.wst)(v) = \bot \end{cases} \\
[\![\kappa]\!]_{\mathrm{btrm}}(\sigma) &:= \lambda(v_1 \in \mathbb{N}_{160}, v_2 \in \mathbb{N}_{160}, f_\bot \in Fid \cup \{\bot\}).(call(v_1, v_2, f_\bot) \in \sigma.cs) \\
[\![u]\!]_{\mathrm{btrm}}(\sigma) &:= (\sigma.\zeta)(u) \\
[\![ths]\!]_{\mathrm{btrm}}(\sigma) &:= \sigma.ee.ths \\
[\![clr]\!]_{\mathrm{btrm}}(\sigma) &:= \sigma.ee.clr \\
[\![cvl]\!]_{\mathrm{btrm}}(\sigma) &:= \sigma.ee.cvl \\
[\![\varepsilon]\!]_{\mathrm{btrm}}(\sigma) &:= \sigma.flg
\end{aligned}
$$

We define the interpretation of terms as follows.

$$
\begin{aligned}
[\![\underline{t}]\!]_{\mathrm{trm}}(\sigma) &:= [\![\underline{t}]\!]_{\mathrm{btrm}}(\sigma) \\
[\![t_1\ aop\ t_2]\!]_{\mathrm{trm}}(\sigma) &:= \begin{cases} [\![aop]\!]([\![t_1]\!]_{\mathrm{trm}}(\sigma), [\![t_2]\!]_{\mathrm{trm}}(\sigma)) \,\%\, |\mathbb{N}_{256}| & \text{if } [\![t_1]\!]_{\mathrm{trm}}(\sigma) \in \mathbb{N} \wedge \\ & \quad [\![t_2]\!]_{\mathrm{trm}}(\sigma) \in \mathbb{N} \\ \bot & \text{otherwise} \end{cases} \\
[\![\overline{c}]\!]_{\mathrm{trm}}(\sigma) &:= \eta(c) \\
[\![\underline{t}(t)]\!]_{\mathrm{trm}}(\sigma) &:= \begin{cases} ([\![\underline{t}]\!]_{\mathrm{btrm}}(\sigma))([\![t]\!]_{\mathrm{trm}}(\sigma)) & \text{if } [\![t]\!]_{\mathrm{trm}}(\sigma) \in dom([\![\underline{t}]\!]_{\mathrm{btrm}}(\sigma)) \\ \bot & \text{otherwise} \end{cases} \\
[\![\underline{t}[t_1 : t_2]]\!]_{\mathrm{trm}}(\sigma) &:= ([\![\underline{t}]\!]_{\mathrm{btrm}}(\sigma))[[\![t_1]\!]_{\mathrm{trm}}(\sigma) \mapsto [\![t_2]\!]_{\mathrm{trm}}(\sigma)] \\
[\![(t_1, t_2, f_\bot)]\!]_{\mathrm{trm}}(\sigma) &:= ([\![t_1]\!]_{\mathrm{trm}}(\sigma), [\![t_2]\!]_{\mathrm{trm}}(\sigma), f_\bot)
\end{aligned}
$$

We define the interpretation of assertions as follows.

$$
\begin{aligned}
[\![t_1\ cop\ t_2]\!]_{\mathrm{asst}}(\sigma) &:= \begin{cases} [\![cop]\!]([\![t_1]\!]_{\mathrm{trm}}(\sigma), [\![t_2]\!]_{\mathrm{trm}}(\sigma)) & \text{if } [\![t_1]\!]_{\mathrm{trm}}(\sigma) \in \mathbb{N} \wedge [\![t_2]\!]_{\mathrm{trm}}(\sigma) \in \mathbb{N} \\ \bot & \text{otherwise} \end{cases} \\
[\![\phi_1\ bop\ \phi_2]\!]_{\mathrm{asst}}(\sigma) &:= \begin{cases} [\![bop]\!]([\![\phi_1]\!]_{\mathrm{asst}}(\sigma), [\![\phi_2]\!]_{\mathrm{asst}}(\sigma)) & \text{if } [\![\phi_1]\!]_{\mathrm{asst}}(\sigma) \in \{tt, ff\} \wedge \\ & \quad [\![\phi_2]\!]_{\mathrm{asst}}(\sigma) \in \{tt, ff\} \\ \bot & \text{otherwise} \end{cases} \\
[\![\exists \underline{t} : \phi]\!]_{\mathrm{asst}}(\sigma) &:= \exists \nu : [\![\phi]\!]_{\mathrm{asst}}(\sigma[\underline{t} \mapsto \nu])
\end{aligned}
$$

We define the simultaneous substitution on basic terms $\underline{t}[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]$ as follows, where $\underline{t}_1, \ldots, \underline{t}_n$ are distinct basic terms, none of which is a value v.

$$
\underline{t}[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] := \begin{cases} t_i & \text{if } \underline{t}_i = \underline{t} \\ \underline{t} & \text{otherwise} \end{cases}
$$

We define the simultaneous substitution on terms $t[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]$ as follows, where $\underline{t}_1, \ldots, \underline{t}_n$ are distinct basic terms, none of which is a value v.

$$
\begin{aligned}
(t_1 \ aop \ t_2)[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] &:= t_1[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] \ aop \ t_2[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] \\
\overline{c}[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] &:= \overline{c} \\
\underline{t}(t)[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] &:= \underline{t}[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n](t[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]) \\
(t[t_1 : t_2])[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] &:= t[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n][t_1[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] : t_2[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]] \\
(t_1, t_2, f_\perp)[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] &:= (t_1[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n], t_2[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n], f_\perp)
\end{aligned}
$$

We define the simultaneous substitution on assertions $\phi[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]$ as follows, where $\underline{t}_1, \ldots, \underline{t}_n$ are distinct basic terms, none of which is a value v.

$$
\begin{aligned}
(t_1 \ cop \ t_2)[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] &:= t_1[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] \ cop \ t_2[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] \\
(\phi_1 \ bop \ \phi_2)[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] &:= \phi_1[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] \ bop \ \phi_2[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] \\
(\exists \underline{t} : \phi)[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n] &:= \begin{cases}
\exists \underline{t} : (\phi[t_{s_1}/\underline{t}_{s_1}, \ldots, t_{s_m}/\underline{t}_{s_m}]) \\
\qquad \text{if } s_1, \ldots, s_m \text{ are all the indices s s.t.} \\
\qquad\quad \underline{t}_s \in \mathit{fvars}(\exists \underline{t} : \phi) \wedge \underline{t}_s \neq t_s \\
\qquad \text{and } \forall j \in \{1, \ldots, m\} : \underline{t} \notin \mathit{vars}(t_{s_j}) \\
\perp \quad \text{otherwise}
\end{cases}
\end{aligned}
$$

B.2 Definition of the Program Logic

We provide the logical judgments and inference rules missing from Section 4.

The logical judgment for functions is

$$I, \Delta \vdash^c fd$$

The only inference rule for functions is

$$
\frac{\begin{array}{c}
\phi_0 = \bigwedge_i (x'_i = d(\theta'_i)) \wedge x' = d(\theta') \\
\Delta(c, f) = ([x_1, \ldots, x_n, x'], (\Phi, \Phi')) \qquad X = \{x_1, \ldots, x_n, x'_1, \ldots, x'_m, x'\} \\
I, \Delta \vdash^c_X \{\Phi \wedge \phi_0 \wedge \neg\varepsilon\} \ S \ \{\Phi''\} \quad \Phi'' \wedge \neg\varepsilon \Rightarrow \Phi' \quad \Phi''' \wedge \neg\varepsilon \Rightarrow I \\
I, \Delta \vdash^c_X \{\exists w : I[w/\mathfrak{b}] \wedge \mathfrak{b}(\overline{c}) = w(\overline{c}) + \mathit{cvl} \wedge \phi_0 \wedge \neg\varepsilon\} \ S \ \{\Phi'''\}
\end{array}}{I, \Delta \vdash^c \mathsf{fun} \ f(x_1 : \theta_1, \ldots, x_n : \theta_n; x'_1 : \theta'_1, \ldots, x'_m : \theta'_m) \ \mathsf{ret} \ x' : \theta' \ \{S\}}
$$

This rule says that for a function definition with body S, if the following can be established, then the function definition can be verified against its specification. The first condition is: under the pre-condition of the function, the fact that the local variables (that are not parameters) and the return variable have their default values, and the fact that there is no exception, the assertion Φ'' that implies the post-condition of the function can be derived as the post-condition of S. The second condition is: under the fact that the invariant of the contract holds after the balance of the contract is increased by cvl, the fact that the local variables (that are not parameters) are zero, and the fact that there is no exception, the assertion Φ''' that implies the invariant of the contract can be derived as the post-condition of S.

The logical judgment for fallback functions is

$$I, \Delta \vdash^{c} \mathit{fbd}$$

The only inference rule for fallback functions is

$$\frac{\begin{array}{c}\Delta(c, \bot) = ([], (\Phi, \Phi')) \qquad X = \{x_1, \ldots, x_n\} \qquad \phi_0 = \bigwedge_i (x_i = d(\theta_i)) \\ I, \Delta \vdash^{c}_{X} \{\Phi \wedge \phi_0 \wedge \neg\varepsilon\}\ S\ \{\Phi''\} \quad \Phi'' \wedge \neg\varepsilon \Rightarrow \Phi' \quad \Phi''' \wedge \neg\varepsilon \Rightarrow I \\ I, \Delta \vdash^{c}_{X} \{\exists w : I[w/\mathfrak{b}] \wedge \mathfrak{b}(\overline{c}) = w(\overline{c}) + \mathit{cvl} \wedge \phi_0 \wedge \neg\varepsilon\}\ S\ \{\Phi'''\}\end{array}}{I, \Delta \vdash^{c} \mathsf{fun}\ (x_1 : \theta_1, \ldots, x_n : \theta_n)\ \{S\}}$$

The explanation for the premises is analogous to those of the rule for function definitions. Note that for a fallback function, the variables listed before the function body are the local variables – the fallback function has no parameters.

The logical judgment for contracts is

$$I, \Delta \vdash \mathit{ctr}$$

The only inference rule for contracts is

$$\frac{\forall i \in \{1, \ldots, n\} : I, \Delta \vdash^{c} \mathit{fd}_i \qquad I, \Delta \vdash^{c} \mathit{fbd}}{I, \Delta \vdash \mathsf{contract}\ c\ \{\ \mathsf{var}\ \mathit{tvs}\ \mathit{fd}_1, \ldots, \mathit{fd}_n\ \mathit{fbd}\ \}}$$

The logical judgment for lists of contracts is

$$\Delta \vdash \mathit{ctrs}$$

The only inference rule for this judgment is

$$\frac{\forall i \in \{1, \ldots, n\} : I_i, \Delta \vdash \mathit{ctr}_i \qquad \{c \mid \exists f_\bot : (c, f_\bot) \in \mathit{dom}(\Delta)\} = \bigcup_i \mathit{cid}(\mathit{ctr}_i)}{\Delta \vdash [\mathit{ctr}_1, \ldots, \mathit{ctr}_n]}$$

B.3 Soundness of the Program Logic

Lemma 3. *The following equality holds as long as both sides of it are defined.*

$$[\![t[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]]\!]_{\mathrm{trm}}(\sigma) = [\![t]\!]_{\mathrm{trm}}(\sigma[\underline{t}_1 \mapsto [\![t_1]\!]_{\mathrm{trm}}(\sigma), \ldots, \underline{t}_n \mapsto [\![t_n]\!]_{\mathrm{trm}}(\sigma)])$$

Proof (Sketch). We implicitly have that $\underline{t}_1, \ldots, \underline{t}_n$ are pairwise distinct from the notation of simultaneous substitution. We proceed by induction on the structure of t. We treat some typical cases.

Case $t = w$. We make a case distinction on whether there is some $j \in \{1, \ldots, n\}$, such that t is the same as $\underline{t}_j$. If there is no such j, then LHS is $[\![w]\!]_{\mathrm{btrm}}(\sigma)$, which is the same as RHS. If there is some j such that $\underline{t}_j = t$, then LHS is $[\![t_j]\!]_{\mathrm{trm}}(\sigma)$, which is again the same value as RHS.

Case $t = t'_1\ aop\ t'_2$. LHS is equal to

$$[\![t'_1[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]\ aop\ t'_2[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]]\!]_{\mathrm{trm}}(\sigma)$$

Using the definition of the interpretation of the arithmetic operations on terms, as well as the IH, this is equal to

$$[\![aop]\!]([\![t'_1]\!]_{\mathrm{trm}}(\sigma[\underline{t}_1 \mapsto [\![t_1]\!]_{\mathrm{trm}}(\sigma), \ldots, \underline{t}_n \mapsto [\![t_n]\!]_{\mathrm{trm}}(\sigma)]),\\ [\![t'_2]\!]_{\mathrm{trm}}(\sigma[\underline{t}_1 \mapsto [\![t_1]\!]_{\mathrm{trm}}(\sigma), \ldots, \underline{t}_n \mapsto [\![t_n]\!]_{\mathrm{trm}}(\sigma)])) \,\%\, |\mathbb{N}_{256}|$$

Using the definition of the arithmetic operations on terms again, we deduce the equality LHS=RHS.

Case $t = \overline{c}$. Both sides are equal to $\eta(c)$. Hence, we have the desired result. □

Lemma 4. *We have the following equality as long as both sides are well-defined.*

$$[\![\phi[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]]\!]_{\mathrm{asst}}(\sigma) = [\![\phi]\!]_{\mathrm{asst}}(\sigma[\underline{t}_1 \mapsto [\![t_1]\!]_{\mathrm{trm}}(\sigma), \ldots, \underline{t}_n \mapsto [\![t_n]\!]_{\mathrm{trm}}(\sigma)])$$

Proof (Sketch). The proof is by induction on the structure of ϕ. The case where $\phi = t_1\ cop\ t_2$ is solved using Lemma 3. The case where $\phi = t_1\ bop\ t_2$ is solved using the IH. We present the details for $\phi = \exists \underline{t} : \phi'$.

Case $\phi = \exists \underline{t} : \phi'$. We have

$$\begin{aligned}
\mathrm{LHS} = {} & [\![(\exists \underline{t} : \phi')[t_1/\underline{t}_1, \ldots, t_n/\underline{t}_n]]\!]_{\mathrm{asst}}(\sigma) \\
= {} & [\![\exists \underline{t} : (\phi'[t_{s_1}/\underline{t}_{s_1}, \ldots, t_{s_m}/\underline{t}_{s_m}])]\!]_{\mathrm{asst}}(\sigma) \quad \text{(using the definition of substitution)} \\
= {} & \exists \nu : [\![\phi'[t_{s_1}/\underline{t}_{s_1}, \ldots, t_{s_m}/\underline{t}_{s_m}]]\!]_{\mathrm{asst}}(\sigma[\underline{t} \mapsto \nu]) \\
= {} & \exists \nu : [\![\phi']\!]_{\mathrm{asst}}(\sigma[\underline{t} \mapsto \nu][\underline{t}_{s_1} \mapsto [\![t_{s_1}]\!]_{\mathrm{trm}}(\sigma[\underline{t} \mapsto \nu]), \ldots, \underline{t}_{s_m} \mapsto [\![t_{s_m}]\!]_{\mathrm{trm}}(\sigma[\underline{t} \mapsto \nu])]) \\
= {} & \exists \nu : [\![\phi']\!]_{\mathrm{asst}}(\sigma[\underline{t} \mapsto \nu][\underline{t}_{s_1} \mapsto [\![t_{s_1}]\!]_{\mathrm{trm}}(\sigma), \ldots, \underline{t}_{s_m} \mapsto [\![t_{s_m}]\!]_{\mathrm{trm}}(\sigma)]) \\
& \quad \text{(because } \forall j \in \{1, \ldots, m\} : \underline{t} \notin \mathit{vs}(\underline{t}_{s_j})\text{)} \\
= {} & \exists \nu : [\![\phi']\!]_{\mathrm{asst}}(\sigma[\underline{t}_{s_1} \mapsto [\![t_{s_1}]\!]_{\mathrm{trm}}(\sigma), \ldots, \underline{t}_{s_m} \mapsto [\![t_{s_m}]\!]_{\mathrm{trm}}(\sigma)][\underline{t} \mapsto \nu]) \\
& \quad \text{(because } \underline{t}_{s_1}, \ldots, \underline{t}_{s_m} \in \mathit{fvs}(\exists \underline{t} : \phi''), \text{ which gives } \underline{t} \neq \underline{t}_{s_1}, \ldots, \underline{t} \neq \underline{t}_{s_m}\text{)}
\end{aligned}$$

On the other hand, we have

$$\begin{aligned}
\mathrm{RHS} = {} & [\![(\exists \underline{t} : \phi')]\!]_{\mathrm{asst}}(\sigma[\underline{t}_1 \mapsto [\![t_1]\!]_{\mathrm{trm}}(\sigma), \ldots, \underline{t}_n \mapsto [\![t_n]\!]_{\mathrm{trm}}(\sigma)]) \\
= {} & \exists \nu : [\![\phi']\!]_{\mathrm{asst}}(\sigma[\underline{t}_1 \mapsto [\![t_1]\!]_{\mathrm{trm}}(\sigma), \ldots, \underline{t}_n \mapsto [\![t_n]\!]_{\mathrm{trm}}(\sigma)][\underline{t} \mapsto \nu])
\end{aligned}$$

It can then be deduced that LHS = RHS, using the facts that each $j \in \{1, \ldots, n\} \setminus \{s_1, \ldots, s_m\}$ satisfies $\underline{t}_j \notin \mathit{fvs}(\exists \underline{t} : \phi')$ or $\underline{t}_j = t_j$, and that $\underline{t}_{s_1}, \ldots, \underline{t}_{s_m}$ are pair-wise distinct, and with a case distinction on whether $\underline{t}$ is $\underline{t}_j$ for some $j \in \{1, \ldots, n\} \setminus \{s_1, \ldots, s_m\}$. □

Lemma 5. *If* $X = \mathit{dom}(\mathit{lst})$, $\eta(c) = \mathit{ee.ths}$, $\sigma = (\mathit{lst}, \mathit{wst}, \mathit{ee}, \mathit{cs}, \mathit{flg}, \zeta)$, *and* $\langle e, \mathit{lst}, \mathit{wst}, \mathit{ee}\rangle \downarrow^{\eta} v : \theta$, *then* $[\![[e]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) = v$.

Proof (Sketch). We proceed by induction on the structure of e. We present two typical cases.

Case $e = x\langle e_1 \rangle$. According to the semantics, we have $\langle e_1, lst, wst, ee\rangle \downarrow^{\eta} v_1 : \theta_1$ for some v_1 and θ_1 such that

$$v_1 \in dom(wst(ee.ths).st(x)) \tag{13}$$

$$v = wst(ee.ths).st(x)(v_1) \tag{14}$$

According to the semantics, we also have $x \notin dom(lst)$. Hence, $x \notin X$. According to the definition of $[\cdot]^{\mathrm{c}}_X$, and the evaluation function for terms, we have $[\![[x\langle e_1\rangle]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) = [\![x^c([e_1]^{\mathrm{c}}_X)]\!]_{\mathrm{trm}}(\sigma) = ([\![x^c]\!]_{\mathrm{trm}}(\sigma))([\![[e_1]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma))$, under the condition $[\![[e_1]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) \in dom([\![x^c]\!]_{\mathrm{trm}}(\sigma))$. Moreover, we have

$$[\![x^c]\!]_{\mathrm{trm}}(\sigma) = wst(\eta(c)).st(x) = wst(ee.ths).st(x)$$

Moreover, we have $[\![[e_1]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) = v_1$ using the IH. Ultimately, we have

$$[\![[x\langle e_1\rangle]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) = wst(ee.ths).st(x)(v_1) = v$$

using (13) and (14).

Case $e = e_1\ aop\ e_2$. We have $v = [\![aop]\!](v_1, v_2)\%|\mathbb{N}_{256}|$, where v_1 is such that $\langle e_1, lst, wst, ee\rangle \downarrow^{\eta} v_1 : \mathsf{U256}$, and $\langle e_2, lst, wst, ee\rangle \downarrow^{\eta} v_2 : \mathsf{U256}$. By the IH, we have $[\![[e_1]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) = v_1$, and $[\![[e_2]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) = v_2$. Hence, by the definition of the interpretation function for terms, we have

$$[\![[e_1\ aop\ e_2]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) = [\![aop]\!]([\![[e_1]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma), [\![[e_2]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma))\%|\mathbb{N}_{256}|$$

Hence, we have $[\![[e_1\ aop\ e_2]^{\mathrm{c}}_X]\!]_{\mathrm{trm}}(\sigma) = v$. □

Lemma 6. *If* $wf_{\mathrm{wst}}(wst)$, $lte \triangleright lst$, $ste = ste\text{-}of(wst(ee.ths).ctr)$, *it can be derived that* $ctrs, ste, lte \vdash S$, *and* $\langle S_1, lst_1, wst_1, ee_1\rangle \rightarrow_{\eta} (lst'_1, wst'_1, cs_1)_{fl g_1}$ *is a node in the derivation tree for* $\langle S, lst, wst, ee\rangle \rightarrow_{\eta} (lst', wst', cs)_{fl g}$, *then we have* $wf_{\mathrm{wst}}(wst_1)$, *and there exist* ste_1, lte_1, *and* $ctrs'$, *such that* $ste_1 = ste\text{-}of(wst_1(ee_1.ths).ctr)$, $lte_1 \triangleright lst_1$, *and* $ctrs', ste_1, lte_1 \vdash S_1$ *can be derived.*

Proof (Sketch). We proceed with an induction on the semantic derivation for S. We give three typical cases.

Case skip. We have that S_1, lte_1, wst_1, and ee_1 agree with S, lte, wst, and ee, respectively. It is straightforward to establish the conclusion of the lemma.

Case while (e) $\{S_0\}$, with

$$\langle S_0, lst, wst, ee\rangle \rightarrow_{\eta} (lst'', wst'', cs_1)_{ff} \tag{15}$$

and

$$\langle \mathsf{while}\ (e)\ \{S_0\}, lst'', wst'', ee\rangle \rightarrow_{\eta} (lst', wst', cs_2)_{fl g} \tag{16}$$

We distinguish between three cases.

– If the node for S_1 is the root of the derivation tree for while (e) $\{S_0\}$, then the reasoning is analogous to the case for skip.

– If the node for S_1 is in the sub-tree for (15), then we derive $ctrs, ste, lte \vdash S_0$ from $ctrs, ste, lte \vdash \mathsf{while}\ (e)\ \{S_0\}$. Then, we establish the desired result using the IH.
– If the node for S_1 is in the sub-tree for (16), then we use Lemma 2 to establish $wf_{\mathrm{wst}}(wst'')$, and $lte \triangleright lst''$. We then use the IH to establish the desired result.

Case $e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x$, with the intended callee found in the target contract. We distinguish between two cases.

– If the node for S_1 is the root of the derivation tree for $e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x$, then the reasoning is analogous to the case for skip.
– If the node for S_1 is in the sub-tree for the evaluation of the callee, represented as $\langle S_0, lst'_0, wst, ee'_0\rangle \to_\eta (lst'', wst'', cs'')_{flg''}$, then we proceed as follows. From $ctrs, ste, lte \vdash e_{\mathrm{a}}.f(e_1, \ldots, e_n) \to x$, we have $ctrs, ste, lte \vdash e_1 : \theta_1, \ldots, ctrs, ste, lte \vdash e_n : \theta_n$, for some $\theta_1, \ldots, \theta_n$. Using Lemma 1, it can be deduced that $v_1 \in [\![\theta_1]\!], \ldots, v_n \in [\![\theta_n]\!]$, where $v_1 : \theta_1$ and $v_n : \theta_n$ are the evaluation results for $e_1, \ldots, e_n$. Hence, it can be seen that $\theta_1, \ldots, \theta_n$ are the types declared for the parameters of the callee function, using the semantics. Let $lte_1 := [x_1 \mapsto \theta_1, \ldots, x_n \mapsto \theta_n, x'_1 \mapsto \theta'_1, \ldots, x'_m \mapsto \theta'_m, x' \mapsto \theta']$, where $\theta'_1, \ldots, \theta'_m$, and θ' are the types declared for the local variables and the return variable of the callee function. We then have $lte_1 \triangleright lst'_0$ because of $v_1 \in [\![\theta_1]\!], \ldots, v_n \in [\![\theta_n]\!]$. Let $ste_1 := ste\text{-}of(wst(ee'_0.ths).ctr)$. We can then establish $ctrs_1, ste_1, lte_1 \vdash S_0$ for some $ctrs_1$ using $wf_{\mathrm{wst}}(wst)$. We then use the IH to establish the desired result.

□

Lemma 7. *If the following conditions are satisfied*

1. $ctrs = [ctr_1, \ldots, ctr_n]$,
2. $\vdash ctrs$,
3. $\Delta \vdash ctrs$,
4. $\exists j \in \{1, \ldots, n\} : ctr_j = ctr$,
5. $cid(ctr) = c$,
6. $ctrs, ste\text{-}of(ctr), lte \vdash S$,
7. $lte \triangleright lst$,
8. $wf_{\mathrm{wst}}(wst)$,
9. $I, \Delta \vdash^{\mathrm{c}}_X \{\phi\}\ S\ \{\phi'\}$,
10. $dom(lst) = X$,
11. $\forall i \in \{1, \ldots, n\} : wst(\eta(cid(ctr_i))).ctr = ctr_i$,
12. $\eta(c) = ee.ths$,
13. $\langle S, lst, wst, ee\rangle \to_\eta (lst', wst', cs)_{flg}$,
14. $[\![\phi]\!]_{\mathrm{asst}}(lst, wst, ee, cs_0, ff, \zeta) = tt$,

then its holds that $[\![\phi']\!]_{\mathrm{asst}}(lst', wst', ee, cs_0 \cup cs, flg, \zeta) = tt$.

Proof (Sketch). Because of the involvement of the subsumption rule in the program logic, we perform an induction on the shape of derivation trees in the program logic for statements. We perform an inner induction on the semantic derivation in the case of the while loop, function calls, and transfers. We present

the relatively involved cases for assignments to mapping elements (updates of mappings), and function calls.

In the following, let

$$\sigma := (lst, wst, ee, cs_0, \mathit{ff}, \zeta)$$
$$\sigma' := (lst', wst', ee, cs_0 \cup cs, \mathit{flg}, \zeta)$$

Case $I, \Delta \vdash^c_X \{\phi\}\ x\langle e_1\rangle := e_2\ \{\phi'\}$. By $ctrs, ste\text{-}of(ctr), lte \vdash S$, we have $x \notin dom(lte)$, and $x \in ste\text{-}of(ctr)$. Hence, $x \notin dom(lst)$, because of $lte \triangleright lst$. Hence, $x \notin X$. Hence, $[x]^c_X = x^c$. Hence, we have

$$[\![\phi']\!]_{\text{asst}}(\sigma') = \exists \nu : [\![\phi[w/x^c] \wedge x^c = w[[e_1]^c_X[w/x^c] : [e_2]^c_X[w/x^c]] \wedge \neg\varepsilon]\!]_{\text{asst}}(\sigma'[w \mapsto \nu])$$

using the definition of the interpretation function for assertions, and the definition of ϕ' in the type system.

We have $wst(ee(c)).ctr = ctr$ using the conditions of the lemma. Hence, we have $ste\text{-}of(ctr) \blacktriangleright wst(\eta(c)).st$ using $wf_{\text{wst}}(wst)$. Let $\nu_0{:=}wst(\eta(c)).st(x)$. Then, ν_0 is a function, as can be seen using $ctrs, ste\text{-}of(ctr), lte \vdash x\langle e_1\rangle := e_2$ and $ste\text{-}of(ctr) \blacktriangleright wst(\eta(c)).st$. According to the semantics, we have

$$\sigma' = \sigma[wst := \sigma.wst[ee.ths \mapsto \sigma.wst(ee.ths)[st := \sigma.wst(ee.ths).st[x \mapsto \nu']]]] \tag{17}$$

for some $ee = \sigma.ee$, and

$$\nu' = \nu_0[v_1 \mapsto v_2] \tag{18}$$

where v_1 and v_2 are such that $\langle [e_1, e_2], lst, wst, ee\rangle \downarrow^\eta [v_1 : \theta_1, v_2 : \theta_2]$. Hence, we have

$$\sigma'[x^c \mapsto \nu_0] = \sigma \tag{19}$$

using the definition of the update of assertion states, and $ee(c) = ee.ths$. With the evaluation of e_1 and e_2, $ctrs, ste\text{-}of(ctr), lte \vdash e_1$, $ctrs, ste\text{-}of(ctr), lte \vdash e_2$, and $wst(ee.ths).ctr = ctr$, we have $[\![[e_1]^c_X]\!]_{\text{trm}}(\sigma) = v_1$, and $[\![[e_2]^c_X]\!]_{\text{trm}}(\sigma) = v_2$, using Lemma 5.

Hence, we have

$$\begin{aligned}
& [\![\phi[w/x^c] \wedge x^c = w[[e_1]^c_X[w/x^c] : [e_2]^c_X[w/x^c]] \wedge \neg\varepsilon]\!]_{\text{asst}}(\sigma'[w \mapsto \nu_0]) \\
= \ & [\![\phi]\!]_{\text{asst}}(\sigma'[x^c \mapsto \nu_0]) \wedge \\
& [\![x^c]\!]_{\text{trm}}(\sigma'[w \mapsto \nu_0]) = \nu_0[[\![[e_1]^c_X]\!]_{\text{trm}}(\sigma'[x^c \mapsto \nu_0]) \mapsto [\![[e_2]^c_X]\!]_{\text{trm}}(\sigma'[x^c \mapsto \nu_0])] \\
& \text{(using Lemma 3, Lemma 4, } \sigma.\mathit{flg} = \mathit{ff}\text{, (19), and } w \notin vs(\phi)) \\
= \ & [\![\phi]\!]_{\text{asst}}(\sigma) \wedge [\![x^c]\!]_{\text{trm}}(\sigma'[w \mapsto \nu_0]) = \nu_0[[\![[e_1]^c_X]\!]_{\text{trm}}(\sigma) \mapsto [\![[e_2]^c_X]\!]_{\text{trm}}(\sigma)] \\
& \text{(using (19))} \\
= \ & tt \ \text{ (using } [\![\phi]\!]_{\text{asst}}(\sigma) = tt\text{, (17), (18), } [\![[e_1]^c_X]\!]_{\text{trm}}(\sigma) = v_1\text{, and } [\![[e_2]^c_X]\!]_{\text{trm}}(\sigma) = v_2)
\end{aligned}$$

This gives $[\![\phi']\!]_{\text{asst}}(\sigma') = tt$.

Case $I, \Delta \vdash^{c}_{X} \{\phi\}\ \mathsf{adr}(c').f(e_1, \ldots, e_n) \to x\ \{\phi'\}$. We have

$$S = \mathsf{adr}(c').f(e_1, \ldots, e_n) \to x$$

Using the hypothesis that $ctrs, ste\text{-}of(ctr), lte \vdash S$, we have

$$ctrs, ste\text{-}of(ctr), lte \vdash e_i : \theta_i$$

for each i, $c' \in cids(ctrs)$, and that there is some function definition

$$\mathsf{fun}\ f(x_1 : \theta_1, \ldots, x_n : \theta_n; \ldots)\ \mathsf{ret}\ x' : \theta'\ \{S'_0\}$$

in some contract ctr' with $cid(ctr') = c'$ in $ctrs$, such that $lte(x) = \theta'$. We have $\langle \mathsf{adr}(c'), lst, wst, ee\rangle \downarrow^{\eta} \eta(c') : \mathsf{U160}$ according to the semantics of expressions. We have $wst(\eta(c')).ctr = ctr'$ using the hypotheses of the lemma. Because $wst(\eta(c')) \neq \bot$, each e_i is evaluated before executing the callee, according to the semantics. Hence, we have $\langle e_i, lst, wst, ee\rangle \downarrow^{\eta} v_i : \theta'_i$ for some v_i and θ'_i. Using the typing of e_i, $wf_{\mathrm{wst}}(wst)$, $lte \triangleright lst$, and Lemma 1, it can be deduced that $ctrs, ste\text{-}of(ctr), lte \vdash (v_i : \theta'_i) : \theta_i$. Hence, $\theta'_i = \theta_i$ and $v_i \in [\![\theta_i]\!]$. Moreover, we have some v' such that $lst(x) = v' : \theta'$ because of $lte\text{-}of(x) = \theta'$ and $lte \triangleright lst$. Hence, the types of the arguments e_i and the return variable x match those in the function definition in ctr'. Hence, the semantic rule for the case where the intended callee function exists in the target contract is used in establishing $\langle S, lst, wst, ee\rangle \to_{\eta} (lst', wst', cs)_{flg}$. Hence, we have

$$\langle S'_0, lst'_0, wst, ee'_0\rangle \to_{\eta} (lst'', wst'', cs)_{flg'} \tag{20}$$

where S'_0 is the body of the callee function, lst'_0 is the initial local state for evaluating S'_0, wst is the initial world state for the overall call, $ee'_0 = (\eta(c'), ee.ths, 0)$, and $flg' \in \{tt, ff\}$. Hence, we have $ee'_0.ths = \eta(c')$. Using $\vdash ctrs$, and the fact that ctr' is in $ctrs$, we can deduce

$$ctrs, ste\text{-}of(ctr'), lte_1 \vdash S'_0 \tag{21}$$

where lte_1 is the function mapping each variable in $dom(lst'_0)$ to its type as recorded in lst'_0, and each variable not in $dom(lst'_0)$ to $\bot$. Hence, we have $lte_1 \triangleright lst'_0$. Because of $\Delta \vdash ctrs$, we have

$$I, \Delta \vdash^{c'}_{X'} \{\Phi\}\ S'_0\ \{\Phi''\} \tag{22}$$

for some I, $X' = dom(lst'_0)$, and $\Phi'' \wedge \neg\varepsilon \Rightarrow \Phi'$, where Φ and Φ' are the pre-condition and post-condition in $\Delta(c', f)$. With $[\![\phi]\!]_{\mathrm{asst}}(\sigma) = tt$ and $\sigma.flg = ff$, we have $[\![\phi \wedge \neg\varepsilon]\!]_{\mathrm{asst}}(\sigma) = tt$. Hence, there exist a list $\vec{\nu}$ of values such that

$$[\![\Phi[[e_1]^{c}_{X}/x_1, \ldots, [e_n]^{c}_{X}/x_n]\delta]\!]_{\mathrm{asst}}(\sigma[lvs(\Phi) \mapsto \vec{\nu}]) = tt$$

according to the premise of the logic rule for $\mathsf{adr}(c').f(e_1, \ldots, e_n) \to x$, and the interpretation of assertions. Using $\langle e_i, lst, wst, ee\rangle \downarrow^{\eta} v_i : \theta_i$, $X = dom(lst)$, and $\eta(c) = ee.ths$, it can be deduced that $[\![[e_i]^{c}_{X}]\!]_{\mathrm{trm}}(\sigma) = v_i$, using Lemma 5.

Since the local variables in the pre-condition Φ are limited to the parameters $x_1, \ldots, x_n$, it can be deduced that

$$[\![\Phi]\!]_{\text{asst}}(lst'_0, wst, ee'_0, cs_0, \mathit{ff}, \zeta'_0) = tt \tag{23}$$

where $\zeta'_0 = \zeta[lvs(\Phi) \mapsto \vec{\nu}]$, using Lemma 4, the definitions of lst'_0 and ee'_0, and the definition of the update of assertion states. Using the conditions of this lemma, $cid(ctr') = c'$, (21), $lte_1 \triangleright lst'_0$, $wf_{\text{wst}}(wst)$, (22), $X' = dom(lst'_0)$, $\eta(c') = ee'_0.ths$, (20), (23), and the IH of the inner induction, we have

$$[\![\Phi'']\!]_{\text{asst}}(lst'', wst'', ee'_0, cs_0 \cup cs, \mathit{flg}', \zeta'_0) = tt \tag{24}$$

We distinguish between two cases, $\mathit{flg}' = tt$ and $\mathit{flg}' = \mathit{ff}$.

- Suppose $\mathit{flg} = tt$. In this case, we have $\sigma' = \sigma[\mathit{flg} := tt]$ according to the semantics. Hence, it can be shown that $[\![\phi[\mathit{ff}/\varepsilon] \wedge \varepsilon]\!]_{\text{asst}}(\sigma') = tt$ using $[\![\phi]\!]_{\text{asst}}(\sigma) = tt$ and Lemma 4. It can then be deduced that $[\![\phi']\!]_{\text{asst}}(\sigma') = tt$.
- Suppose $\mathit{flg} = \mathit{ff}$. In this case, it can be shown that $[\![\phi'_0 \wedge \neg\varepsilon]\!]_{\text{asst}}(\sigma') = tt$, where ϕ'_0 is the ϕ from the post-condition of the logic rule for the function call $\mathsf{adr}(c').f(e_1, \ldots, e_n) \to x$. Instantiating the logical variables in $lvs(\Phi)$ with the corresponding values in $\vec{\nu}$, w with $lst''(x')$, and w' with $\lambda(v_1, v_2, f_\perp).call(v_1, v_2, f_\perp) \in cs_0 \cup cs$, we can deduce that $\Phi''[w/x'][w'/\kappa]\delta \wedge x = w \wedge \kappa = w'[(\overline{c}, \overline{c'}, f) : tt]$ is satisfied in the post-state of the overall call, where $\delta = [\overline{c'}/ths][\overline{c}/clr][0/cvl]$. Specifically, we have

$$\begin{aligned}
&[\![\Phi''[w/x'][w'/\kappa]\delta]\!] \\
&\qquad (lst[x \mapsto lst''(x')], wst'', ee, cs_0 \cup cs \cup \{call(ee.ths, v, f)\}, \mathit{ff}, \zeta) \\
&\qquad [lvs(\Phi) \mapsto \vec{\nu}][w \mapsto lst''(x')][w' \mapsto \lambda(v_1, v_2, f_\perp).call(v_1, v_2, f_\perp) \in cs_0 \cup cs] \\
=&[\![\Phi'']\!](lst[x \mapsto lst''(x')], wst'', ee'_0, cs_0 \cup cs \cup \{call(ee.ths, v, f)\}, \mathit{ff}, \zeta) \\
&\qquad [lvs(\Phi) \mapsto \vec{\nu}][x' \mapsto lst''(x')][\kappa \mapsto \lambda(v_1, v_2, f_\perp).call(v_1, v_2, f_\perp) \in cs_0 \cup cs] \\
=&[\![\Phi'']\!](lst[x \mapsto lst''(x')][x' \mapsto lst''(x')], wst'', ee'_0, cs_0 \cup cs, \mathit{ff}, \zeta'_0) \\
=&tt
\end{aligned}$$

In the derivation above, we have used Lemma 4, the definition of the update of assertion states, the condition $vs(\Phi'') \cap \{w, w'\} = \emptyset$, the fact that $\zeta'_0 = \zeta[lvs(\Phi) \mapsto \vec{\nu}]$, (24), and the global assumption that the local variables in the post-condition Φ'' can only be the return variable x'. By instantiating the existentially quantified $\mathfrak{b}$, κ, the state variables in $svs(\phi, \Phi, [e_1]^c_X, \ldots, [e_n]^c_X)$, and x with their values immediately before the call, it can also be verified that the remaining part of ϕ'_0 is satisfied in the post-state of the call.

Case $\nabla, \Delta \vdash^c_X \{\phi\}\ e_{\text{a}}.f(e_1, \ldots, e_n) \to x\ \{\phi'\}$. We make a case analysis on whether there are actions of the form $call(v_0, \eta(c), f_\perp)$ for some v_0 in the set $\sigma'.cs \setminus \sigma.cs$.

- If there is no action of the form $call(v_0, \eta(c), f_\perp)$ in $\sigma'.cs \setminus \sigma.cs$ for any v_0, then the execution of the call $e_{\text{a}}.f(e_1, \ldots, e_n) \to x$ ends with $(lst, wst, \emptyset)_{tt}$

or there is no successful reentrance into the contract ctr in the execution of the call. In the former case, we have

$$[\![\phi[\mathit{ff}/\varepsilon] \wedge \varepsilon]\!]_{\text{asst}}(\sigma') = tt$$

using $[\![\phi]\!]_{\text{asst}}(\sigma) = tt$, Lemma 4, and the definition of σ'. Hence, we have $[\![\phi']\!]_{\text{asst}}(lst', wst', ee, cs_0 \cup cs, \mathit{flg}, \zeta) = tt$, because $\phi[\mathit{ff}/\varepsilon] \wedge \varepsilon$ is a disjunct in ϕ'. In the latter case, let ϕ'_0 be the ϕ' in the logic rule for $e_{\text{a}}.f(e_1, \ldots, e_n) \rightarrow x$. It can be shown that the first disjunct of ϕ'_0 has the interpretation result tt in σ'. This is by instantiating $\exists w$ with $\lambda v.(wst(v).bal)$, instantiating $\exists w'$ with $\lambda(v_1, v_2, f_\perp).(call(v_1, v, f_\perp) \in cs_0)$, instantiating each state variable in $\exists svs(\phi) \setminus \{x^c \mid x \in Var\}$ with the value of the variable in the initial world state wst for the call, and instantiating $\exists x$ with the value of x in the initial local state lst for the call.

– If there are actions of the form $call(v_0, \eta(c), f_\perp)$ in $\sigma'.cs \setminus \sigma.cs$ for some v_0, then there are a number r of disjoint sub-trees in the semantic derivation for the function call, such that each sub-tree corresponds to the execution of the current contract due to reentrance. We have $[\![\phi \wedge \neg\varepsilon]\!]_{\text{asst}}(\sigma) = tt$ using $[\![\phi]\!]_{\text{asst}}(\sigma) = tt$ and $\sigma.\mathit{flg} = \mathit{ff}$. Hence, $[\![\exists lvs(I) : I]\!]_{\text{asst}}(\sigma) = tt$ according to the premise of the logic rule for $e_{\text{a}}.f(e_1, \ldots, e_n) \rightarrow x$. Hence,

$$[\![I]\!]_{\text{asst}}(\sigma[lvs(I) \mapsto \vec{\nu}]) = tt \tag{25}$$

for some list $\vec{\nu}$ of values (which can be functions). Suppose the first sub-tree corresponding to a successful reentrance has

$$\langle S_1, lst_1, wst_1, ee_1\rangle \rightarrow_\eta (lst'_1, wst'_1, cs'_1)_{\mathit{flg}_1} \tag{26}$$

as its root for some cs'_1 and flg_1, where $S_1 = \mathit{stmt\text{-}of}(ctr, f_{\perp 1})$ for some $f_{\perp 1}$ (i.e., S_1 is the body of the function with identifier $f_{\perp 1}$ in the contract ctr), lst_1 and lst'_1 are the initial and final local states, respectively, for the reentrant call to $f_{\perp 1}$, wst_1 and wst'_1 are the initial and final world states, respective, for this reentrant call, and ee_1 is the execution environment for S_1. Hence, using $\Delta \vdash ctrs$ we obtain

$$I, \Delta \vdash^{\text{c}}_{X_1} \{\exists w : I[w/\mathfrak{b}] \wedge \mathfrak{b}(\overline{c}) = w(\overline{c}) + cvl \wedge \phi_0 \wedge \neg\varepsilon\}\ S_1\ \{\Phi'''\} \tag{27}$$

where $X_1 = dom(lst_1)$, and ϕ_0 is the formula stating that all the local variables and the return variable of the function identified by $f_{\perp 1}$ have their default values, such that $\Phi''' \wedge \neg\varepsilon \Rightarrow I$. Using Lemma 6, it can be deduced that $wf_{\text{wst}}(wst_1)$, and there exist $ctrs_1$ and lte_1, such that $lte_1 \triangleright lst_1$, and $ctrs_1, \mathit{ste\text{-}of}(ctr), lte_1 \vdash S_1$. Using $\vdash ctrs$, ctr is in $ctrs$, and $S_1 = \mathit{stmt\text{-}of}(ctr, f_{\perp 1})$, it can be deduced that $ctrs, \mathit{ste\text{-}of}(ctr), \mathit{lte\text{-}of}(ctr, f_{\perp 1}) \vdash S_1$. Moreover, using the definition of lst_1, we have $\mathit{lte\text{-}of}(ctr, f_{\perp 1}) = lte_1$. Hence, we have

$$ctrs, \mathit{ste\text{-}of}(ctr), lte_1 \vdash S_1 \tag{28}$$

Suppose the parent node of (26) in the derivation tree for $\langle S, lst, wst, ee\rangle \rightarrow_\eta (lst', wst', cs)_{\mathit{flg}}$ is $\langle S_{10}, lst_{10}, wst_{10}, ee_{10}\rangle \rightarrow_\eta (lst'_{10}, wst'_{10}, cs'_{10})_{\mathit{flg}_{10}}$ where

S_{10} is the call (or transfer) statement that triggers the reentrant call to the function $f_{\perp 1}$. It can be shown that $wst_{10}(\eta(c)) = wst(\eta(c))$ using the semantics of statements. Hence, with (25) and the syntactical restrictions on the invariant I, it can be shown that

$$[\![I]\!]_{\text{asst}}(lst_{10}, wst_{10}, ee_{10}, cs_0 \cup cs_1, \mathit{ff}, \zeta_0') = tt$$

where cs_1 is the set of successful calls before the re-entrant call, and $\zeta_0' = \zeta[lvs(I) \mapsto \vec{\nu}]$. More concretely, cs_1 is the union of all sets $\underline{cs}$ such that $\langle \underline{S}, \underline{lst}, \underline{wst}, \underline{ee}\rangle \rightarrow_\eta (\underline{lst'}, \underline{wst'}, \underline{cs})_{\underline{\mathit{flg}}}$ is the root of a largest subtree that is completely traversed before visiting (26) in a depth-first traversal of the derivation tree for $\langle S, lst, wst, ee\rangle \rightarrow_\eta (lst', wst', cs)_{\mathit{flg}}$. We then have

$$[\![\exists w : I[w/\mathfrak{b}] \wedge \mathfrak{b}(\overline{c}) = w(\overline{c}) + cvl \wedge \phi_0 \wedge \neg\varepsilon]\!](lst_1, wst_1, ee_1, cs_0 \cup cs_1, \mathit{ff}, \zeta_0') = tt \tag{29}$$

using the syntactical restrictions on the invariant I and $wst_{10}(\eta(c)) = wst(\eta(c))$. Using the conditions of this lemma, (28), $lte_1 \triangleright lst_1$, $wf_{\text{wst}}(wst_1)$, (27), (26), $X_1 = dom(lst_1)$, and the IH of the inner induction, we have

$$[\![\Phi''']\!]_{\text{asst}}(lst_1', wst_1', ee_1, cs_0 \cup cs_1 \cup cs_1', \mathit{ff}, \zeta_0') = tt$$

Hence, we have $[\![I]\!]_{\text{asst}}(lst_1', wst_1', ee_1, cs_0 \cup cs_1 \cup cs_1', \mathit{ff}, \zeta_0') = tt$ using $\Phi''' \wedge \neg\varepsilon \Rightarrow I$. By induction on the number of subtrees representing the reentrance into ctr, we show that $[\![I]\!]_{\text{asst}}(lst_r', wst_r', ee_r, cs_0 \cup cs_r \cup cs_r', \mathit{ff}, \zeta_0') = tt$, where r is the index of the last sub-tree corresponding to a successful re-entrance. Using the syntactical constraints on the invariant I, we then deduce that

$$[\![I]\!]_{\text{asst}}(lst'', wst'', ee, cs_0 \cup cs', \mathit{ff}, \zeta_0') = tt$$

where cs' is the set of successful calls performed in the overall execution of the callee for the call to $e_{\text{a}}.f(e_1, \ldots, e_n) \rightarrow x$. Thus, instantiating $\exists lvs(I)$ with $\vec{\nu}$, $\exists w$ with the value of $[e_{\text{a}}]^{\text{c}}_X$ in σ, and $\exists w'$ with $\lambda(v_1, v_2, f_\perp).(call(v_1, v_2, f_\perp) \in (cs_0 \cup cs'))$, the second disjunct of ϕ_0' in the logic rule for $e_{\text{a}}.f(e_1, \ldots, e_n) \rightarrow x$ is interpreted to tt in σ'.

The reasoning for the cases of transfers follows analogous principles the cases of the calls as presented above. The cases for the control flow constructs are largely standard. For some of the cases involving substitutions $[tt/\varepsilon]$ and $[\mathit{ff}/\varepsilon]$, Lemma 4 is used. □

Using Lemma 7, Theorem 2 can be established as follows.

Proof of Theorem 2 We assume the following conditions

$$\vdash ctrs \tag{30}$$

$$\Delta \vdash ctrs \tag{31}$$

$$ctr \text{ is in } ctrs \tag{32}$$

$$\exists xs : \Delta(ctr, f_\perp) = (xs, (\Phi, \Phi')) \tag{33}$$

$$wf(ctr, f_\perp, lst, wst) \tag{34}$$

$$\forall x \in nprms\text{-}of(ctr, f_\perp) : \exists \theta : lst(x) = d(\theta) : \theta \tag{35}$$

$$\eta(cid(ctr)) = ee.ths \wedge \forall i : wst(\eta(cid(ctrs!i))).ctr = ctrs!i \tag{36}$$

$$\langle stmt\text{-}of(ctr, f_\perp), lst, wst, ee\rangle \rightarrow_\eta (lst', wst', cs)_{\mathit{ff}} \tag{37}$$

$$[\![\Phi]\!]_{\text{asst}}(lst, wst, ee, cs_0, \mathit{ff}, \zeta) = tt \tag{38}$$

We proceed to show $[\![\Phi']\!]_{\text{asst}}(lst', wst', ee, cs_0 \cup cs, \mathit{ff}, \zeta) = tt$.

We have

$$ctrs = [ctr_1, \ldots, ctr_n] \tag{39}$$

for some n, because $ctrs$ is a list of contracts. Because of (32), we have

$$\exists j \in \{1, \ldots, n\} : ctr_j = ctr \tag{40}$$

Let $c{:=}cid(ctr)$. We then have $c = cid(ctr)$. Let $S{:=}stmt\text{-}of(ctr, f_\perp)$. We then have

$$ctrs, ste\text{-}of(ctr), lte\text{-}of(ctr, f_\perp) \vdash S \tag{41}$$

using (30). Using (34), we have

$$lte\text{-}of(ctr, f_\perp) \triangleright lst \tag{42}$$

We have $wf_{\text{wst}}(wst)$ because of (34). Because of (31), (33), and the definition of S, we have

$$I, \Delta \vdash^c_X \{\Phi \wedge \phi_0 \wedge \neg\varepsilon\}\ S\ \{\Phi''\} \tag{43}$$

where X is the set of local variables (incl. parameters and return variable) of the function identified by $f_\perp$, ϕ_0 is the formula stating that each local variable of this function has its default value, and $\Phi'' \wedge \neg\varepsilon \Rightarrow \Phi'$. Hence, we have $dom(lst) = X$ using (42). We have

$$\forall i \in \{1, \ldots, n\} : wst(\eta(cid(ctr_i))).ctr = ctr_i \tag{44}$$

using (36) and (39). We have $\eta(c) = ee.ths$ using (36) and $c = cid(ctr)$. We have

$$\langle S, lst, wst, ee\rangle \rightarrow_\eta (lst', wst', cs)_{\mathit{ff}} \tag{45}$$

because of (37) and the definition of S. We have

$$[\![\Phi \wedge \phi_0 \wedge \neg\varepsilon]\!]_{\text{asst}}(lst, wst, ee, cs_0, \mathit{ff}, \zeta) = tt \tag{46}$$

using (38), (35), the definition of ϕ_0 and the definition of the interpretation of assertions. Using Lemma 7, (39), (30), (31), (40) , $c = cid(ctr)$, (41), (42), $wf_{\text{wst}}(wst)$, (43), $dom(lst) = X$, (44), $\eta(c) = ee.ths$, (45), and (46), we obtain

$$[\![\Phi'']\!]_{\text{asst}}(lst', wst', ee, cs_0 \cup cs, \mathit{ff}, \zeta) = tt$$

We have $[\![\neg\varepsilon]\!]_{\text{asst}}(lst', wst', ee, cs_0 \cup cs, \mathit{ff}, \zeta) = tt$ using the definition of the interpretation of assertions. Thus, we have

$$[\![\Phi']\!]_{\text{asst}}(lst', wst', ee, cs_0 \cup cs, \mathit{ff}, \zeta) = tt$$

using $\Phi'' \wedge \neg\varepsilon \Rightarrow \Phi'$. □