Is Solidity solid enough?

Silvia Crafa[1], Matteo Di Pirro[1], and Elena Zucca[2]

[1] University of Padova, Italy
[2] DIBRIS, University of Genova, Italy

Abstract. We introduce Featherweight Solidity, a calculus formalizing the core features of the Solidity language, thus providing a fundamental step to reason about safety properties of smart contracts' source code. The formalization includes a static type system that represents the foundation of the Solidity compiler. We show that it prevents some errors whereas many others, such as accesses to a non existing function or state variable, are only detected at runtime and cause interruption and rolling-back of transactions. We then propose a refinement of the type system, that is retro-compatible with original Solidity code and statically captures more errors, such as unsafe casts and unsafe call-back expressions.

Keywords: type soundness · operational semantics · smart contracts

1 Introduction

Smart contracts and their decentralized algorithmic validation are emerging as a successful technology to implement agreements between mutually untrusted parties without relying on a centralized third authority. They are currently used in many critical domains such as infrastructural systems and financial applications, therefore it is of paramount importance to study their correctness and safety properties. In this work we address this problem at the programming language abstraction level, so to *statically rule out harmful patterns* appearing in smart contracts code and *support a safer programming discipline.* More precisely, we focus on Solidity, the most widely used programming language in Ethereum's ecosystem, and its type system, that is integrated in the language so to let the compiler statically enforce basic safety properties of smart contracts.

Our first contribution is the formalization of the semantics of the core of the Solidity language, that we call Featherweight Solidity (FS). The FS calculus focuses on contract instantiation, typed interaction among deployed contracts and money transfers. Even if important features like *gas* fees are omitted, the calculus provides a rather compact and clean model of key aspects of smart contract programming. Such a formalization indeed allows one to precisely define the behavior of many Solidity programs, so to describe undesired behaviors and to investigate on a way to prevent them. This is a fundamental step for the development of many analysis techniques that take advantage of formal methods to verify and to reason about the safety properties of smart contracts' source code, rather than acting at the level of EVM bytecode. Moreover, the formalization style of FS intentionally highlights the connection between objects and smart contracts, opening the way to adapt the rich (typed) theory of OOLs in the context of Solidity.

As a second contribution, we study the type system of FS, in order to clarify, precisely state and most importantly prove, Solidity's claim to be a type-safe language. In particular, we show that the Solidity static type system only prevents stuck executions, but not runtime type errors, such as accesses to a non existing function or state variable. That is, well-typed FS programs, hence also compiled Solidity contracts, may reach specific exceptional states that cause the current transaction to be interrupted and rolled-back, possibly leading to Ether indefinitely locked into a contract's balance. Reverting an unsafe transaction guarantees the consistency of the blockchain, but the account that issued the transaction is not reimbursed for the money it paid to the miner node. Thus, it is of interest of anyone to issue a transaction only when there is a static guarantee that such a transaction will not evolve to a revert.

The main reason for the weakness of Solidity's type safety lies in the fact that the code of contract functions can refer to contract instances only through their public addresses, but the Solidity `address` type is essentially an untyped pointer, which is notoriously a very flexible but subtle feature.

Our third contribution is then a proposal for a refinement of the type system. We show that the enriched type system enjoys a more powerful soundness property, so that the only possible runtime errors in FS remain those due to a negative account balance. In particular, cast expressions or money transfers that would lead to unsafe usage of contract members or calls to an undefined fallback function are now ruled out at compile-time. Moreover, we show that such a refinement can be actually made retro-compatible with original Solidity code. Hence, it is possible for contracts written in the extended safer language to interact with already deployed smart contracts.

The key idea is twofold: first, we refine the `address` type with type information about the contract it refers to. Secondly, we enrich the functions' signatures so to allow functions to be called only by contracts whose address has an expected (super-)type. This additional infomation is particularly useful within contracts code to type check the implicit sender parameter, therefore, besides statically preventing runtime errors, the refined compiler statically prevents unsafe callback expressions, that are notoriously vulnerable Solidity programming patterns. To take advantage of the full power of the refined typing, the major effort required to Solidity programmers is to explicitly express the type constraint they require on contracts callers. However, this requirement actually supports a safer programming discipline, and we put forward a number of convenient function modifiers, in line with Solidity language style, so to enhance the use of its compiler as a convenient building tool.

A preliminary version of FS has been given in [7]. The formalization presented in this paper is cleaner and focused on key aspects; moreover, some subtle Solidity aspects have been modeled in a closer way.

```
contract Bank {
  mapping (address => uint) amounts;
  function withdraw(uint n) {
    require(amounts[msg.sender] >= n);
    amounts[msg.sender] -= n;
    msg.sender.transfer(n);
  }
  function deposit() payable {
     amounts[msg.sender] += msg.value;
  }
}
```

Fig. 1. A simple Bank contract in Solidity

2 Background

Ethereum [6] is a decentralized platform that runs programs called *smart contracts*. Contract instances deployed on the Etherum blockchain are autonomous agents reminiscent of class-based objects in distributed OOLs. They are identified by a unique public address, hold an amount of virtual coins called Ether (*balance*), are given a persistent area in the blockchain where their *state* is stored, and are associated with their immutable executable code. Besides contracts, the blockchain also hosts *Externally Owned Accounts* (EOAs), that correspond to human agents registered to the Ethereum platform. Analogously to smart contracts, EOAs are identified by a unique address and hold an amount of Ether as their balance, but they have no associated code. EOAs start programs by issuing a *transaction*, which either deploys a new contract instance, or invokes a function by sending a message to a contract stored at a given address. Typically, transactions include input data for the invocation, the address of the sender EOA, an amount of virtual money to be transferred to the contract as a sort of payment, and a fee (*gas*) to reward the *miner* node that executes the transaction.

While EOA's initial transactions are written using one of the many API available in Ethereum ecosystem, smart contracts code is commonly written using the Solidity programming language [2], and it is compiled into bytecode running on the Ethereum Virtual Machine (EVM) [15]. As in OOLs, Solidity contracts contain *state variables* and *functions* that, as objects methods, can refer to the currently executing contract instance through the variable `this`. Contract functions can send messages to other (or to the current) contracts, possibly also specifying an amount of virtual money and the gas fee to be paid. Therefore, besides `this`, in Solidity the contract functions have access to the implicit variable `msg`, which stores various information about the current message call, such as the address of the caller (`msg.sender`) and the amount of money sent along with the call (`msg.value`).

As an example, Fig.1 shows a Solidity smart contract that implements a very simple bank. The `amounts` state variable is a mapping that records the amounts of money deposited by clients, either EOAs or smart contracts, indexed by their

Ethereum addresses. To withdraw money from a `Bank` instance `b`, the invocation of the corresponding function should have the standard shape `b.withdraw(n)`. The function body first of all checks whether the caller's bank account contains enough money. If not, an exception (`revert`) is thrown by the runtime and the current transaction is rolled-back, leaving the blockchain as if it had never run. If the caller has enough money, an amount `n` of Wei (Ether's smallest sub-currency) is transferred from the balance of `b` to the balance of the caller by explicitly using the `transfer` primitive. Whenever the caller is a contract, the EVM requires that contract to contain a definition for the so-called *fallback* function, otherwise a `revert` is thrown and the transaction is reverted. The typical purpose of such function is either to track the reception of Ether or to refuse it by throwing an exception. On the contrary, when the recipient of the transfer is an EOA, no fallback function is needed.

The invocation of the `deposit` function, instead, can have the special shape `b.deposit.value(n)()`, binding `n` to the implicit parameter `msg.value`. If not specified, `n` is assumed to be 0. As a consequence, the amount `n` of money is transferred from the balance of the caller (`msg.sender`) to the balance of the `Bank` instance; in this case, no explicit invocation of `transfer` is needed. The state variable `amounts` is updated accordingly. Analogously to the case above, the caller must hold enough money in its balance. Moreover, the additional `value` argument can only be specified for a function with the `payable` modifier, meaning that it is allowed to receive Ether as part of the invocation, otherwise a `revert` would be thrown at invocation time.

3 The Featherweight Solidity calculus

In this section we introduce Featherweight Solidity (FS), a calculus formalizing the core of the Solidity programming language. Many features are omitted, like low level message calls (using the primitives `send, call, delegatecall, callcode`), expressive value types like mappings and first-class function values, function modifiers and multiple inheritance. Furthermore, FS models single transactions, thus it does not deal with the concepts of blocks, distributed block validation, and roll-back of the changes to the blockchain caused by a reverted transaction. We also do not model the concept of *gas* fees, which is a mechanism Ethereum uses to make sure that every transaction eventually terminates and to prevent denial of service attacks.

In such way, we can focus on key aspects of smart contract programming, such as contract instantiation, interactions among deployed contracts and money transfers, and provide a rather compact and clean model of such features. In particular, the definition of FS is inspired by Featherweight Java (FJ) [10], the reference calculus for Java-like languages, exploiting the similarities and highlighting the differences between the notions of object and smart contract. Therefore it opens the way to reuse and adapt the rich and well-known (typed) theory of OOLs in the context of smart contracts.

CT	$::= cds$	contract table
cd	$::=$ `contract` C `is` D $\{sds\ fds\}$	contract declaration
sd	$::= T\ s$`;`	state variable decl.
fd	$::= T\ f$ `(`$T_1\, x_1, \ldots, T_n\, x_n$`){ return` e`; }`	function declaration
e	$::= x \mid$ `u` $\mid$ `n` $\mid a \mid e.s \mid e.s$`=`$e' \mid$ `{`$T\ x$`=`e`;`e'`}` $\mid C(e) \mid$ `revert`$[\lambda]$	expression
	$\mid e.f$`.value(`e^{v}`).sender(`e^{s}`)(`es`)`	
	$\mid$ `new` C`.value(`e^{v}`).sender(`e^{s}`)(`es`)`	
	$\mid$ `address(`e`)` $\mid$ `balance(`e`)` $\mid e$`.transfer(`e^{v}`).sender(`e^{s}`)`	
λ	$::=$ `neg` $\mid$ `rte`	revert label
T	$::= C \mid$ `unit` $\mid$ `uint` $\mid$ `address`	type

Fig. 2. FS: syntax

Syntax and types Syntax and types are given in Fig.2. We assume sets of *variables* x, y, *contract names* C, D, *state variable names* s, *function names* f, *addresses* a. We assume three special variables `this`, `msg.value`, and `msg.sender`, a special contract name `Top`, and a special function name `fb`, all explained below. We adopt the convention that a metavariable which ends by s is implicitly defined as a (possibly empty) sequence, for example cds is defined by $cds ::= \epsilon \mid cd\ cds$, where ϵ denotes the empty sequence.

A contract table is a sequence of contract declarations, consisting of contract name, parent contract's name, a sequence of state variable declarations and a sequence of function declarations. We only model single inheritance and assume a distinguished contract name `Top` with no state variables and function definitions. A function declaration consists of a return type, a function name, a list of typed parameters, and a body which is an expression. Since FS does not model Solidity's function modifiers, every function is implicitly marked `payable` and `external`, that is, can receive Wei and can be invoked by EOAs' transactions. We assume a special function name `fb`, which models the *fallback* function, implicitly invoked whenever money is transferred by means of a `transfer` call. Therefore, if present in a contract definition, the function `fb` must be necessarily declared as `unit fb (){ return` e`; }`. As in FJ, we assume for each contract declaration a canonical constructor.

Expressions includes variables, the only constant `u` of type `unit`, natural constants `n` of type `uint`, addresses, used to refer to EOAs and contracts already deployed in the blockchain, access and assignment to a state variable, and block consisting of a local variable declaration and a body. We use e`;`e' as an abbreviation for `{`$T\ x$`=`e`;`e'`}` with x not free in e'.

The expression $e.f$`.value(`e^{v}`).sender(`e^{s}`)(`es`)` invokes the function f on the contract instance denoted by e, specifying the address e^{s} of the contract instance (or the EOA) that invoked the function, and the amount e^{v} of Wei sent along with the call. In the instantiation expression `new` C`.value(`e^{v}`).sender(`e^{s}`)(`es`)`, the two additional arguments have an analogous meaning.

Assuming that e evaluates to a contract instance, `address(`e`)` returns its address, while, assuming that e evaluates to an address, `balance(`e`)` returns its current balance, and the cast expression $C(e)$ returns the corresponding

```
contract Bank {
  mapping (address => uint) amounts;
  unit deposit() {
    return this.amounts[msg.sender] += msg.value; u
  }
  unit withdraw(uint n) {
    return if this.amounts[msg.sender] >= n
           then this.amounts[msg.sender] -= n;
                msg.sender.transfer(n); u
           else revert
  }
}
```

Fig. 3. A simple Bank contract in FS

contract instance. The expression e`.transfer(`e^{v}`).sender(`e^{s}`)`, assuming that e evaluates to an address, transfers the amount of Wei denoted by e^{v} from the balance of e^{s} to its balance. Finally, the `revert` expression aborts the current transaction. For the aims of our formalization, we add a label λ describing the specific error (`neg` when a money transfer would make an account's balance negative, `rte` for a runtime type error), omitted when not significant.

For simplicity, FS expressions model both Solidity code, that is, smart contracts code, and external code issuing the initial transactions. However, only the latter requires an explicit sender argument in function calls, contract instantiation and money transfer, whereas, in contracts code (that is, in function bodies rather than at top level), the (implicit) sender is always the currently executing contract instance. Formally, we assume that function calls occurring in function bodies have shape $e.f$`.value(`e^{v}`).sender(address(this))(`es`)`, abbreviated $e.f$`.value(`e^{v}`)(`es`)`, and analogously for constructor invocations and transfer.

The syntax of types includes contract names, the `unit` type, the type `uint` of unsigned integers, and the type of addresses. In the definition of FS we privileged uniformity, therefore a FS program is not an executable Solidity program, for instance, Solidity has no `unit` type. However, the correspondence is very close. In Fig.3 we show[3] the FS code corresponding to the Solidity smart contract in Fig.1. As an example, `Bank('0x84b').deposit.value(`500`).sender('0xu7e')()` denotes a transaction issued by the EOA with address `'0xu7e'` to interact with an instance of the `Bank` contract stored at address `'0x84b'`.

Operational semantics The top section of Fig.4 defines runtime expressions, configurations, values and evaluation contexts. Runtime expressions include, besides source-level constructs in Fig.2, *contract references* ι_D^C, where C, D are contract names. We write ι_D as abbreviation for ι_D^D, and omit both when not relevant. When a contract D is instantiated, a new reference ι_D is created with its contract's name built in (as subscript). When a cast to type C occurs at runtime, no

[3] In the examples, we use additional constructs, such as loops, booleans and key-value mappings (for the standard formalization see [7]).

type check is performed by the EVM, but the execution proceeds by recording the target type (as superscript) in the contract reference. That is, ι^C_D is a reference to an instance of contract D (for "dynamic type") that has been cast (that is, statically typed) to type C (for "cast type"). Note that a contract reference keeps its dynamic type forever, whereas it can be used with different cast types. *Configurations*, ranged over by c, are pairs $\langle e, \beta\rangle$, where e is the expression to be evaluated and β the *blockchain*, that stores the global state of the system. Formally, β is finite map from *contract instances* of shape $\langle \iota, a\rangle$ to pairs $\langle vs, \mathsf{n}\rangle$, where n and vs are the contract's *balance* and *state*, respectively, the latter being the tuple of the current values of the state variables. Note that, while the reference records type information, the public address provides an "untyped" way to access a contract instance. As in Ethereum,we assume a one-to-one correspondence between references and addresses in the domain of β, so we can safely use the notations $\beta(\iota)$ and $\beta(a)$ as abbreviations for $\beta(\langle \iota, a\rangle)$. Since Ethereum blockchain records also EOAs addresses and balances, in order to let the map β uniformly deal with both smart contracts and EOAs, we assume that each EOA has a corresponding reference to an instance of a dummy contract `EOContract` whose code only contains a fallback function `fb` with empty body (i.e., `unit fb {return u;}`). Values are contract references and constants of the other types. Evaluation contexts formalize standard left-to-right evaluation (for brevity we do not explicitly list all cases).

The small-step reduction relation over configurations $\longrightarrow_{CT}$ is parameterized by a contract table, omitted to lighten the notation. The reduction rules are collected in Fig.4, where we use the following notations, whose trivial formal definition is omitted. Given a blockchain β: in $\beta[\iota.i{=}v]$, the value of the i-th state variable of the contract instance ι has been replaced by v; in $\beta[\langle \iota, a\rangle \leftarrow vs]$, a new contract instance $\langle \iota, a\rangle$ has been added with state vs and balance 0; in $\beta[a.\mathsf{v} \overset{\mathsf{n}}{\rightsquigarrow} a'.\mathsf{v}]$, an amount of n Wei has been transferred from the balance of the contract instance at address a to that at a'. If $\beta(\langle \iota, a\rangle) = \langle v_1 \dots v_n, \mathsf{n}\rangle$, then we write $\beta(\iota).i$ to denote v_i, for $i \in 1..n$, and we write $\beta(a).\mathsf{v}$ to denote n. The expression $e[v/x]$ is obtained from e by replacing all occurrences of x by v. We write $e[v_1 \dots v_n/x_1 \dots x_n]$ for $e[v_1/x_1] \dots [v_n/x_n]$. The following auxiliary functions are implicitly parameterized on the contract table: $\mathsf{svar}(C, s)$ and $\mathsf{svartype}(C, s)$ return the index and type, respectively, of the state variable s in C, if any; $\mathsf{svars}(C)$ returns the sequence of all (inherited and directly declared) state variables of C; $\mathsf{ftype}(C, f)$ and $\mathsf{fbody}(C, f)$ return the *function type*, of shape $\langle T, T_1 \dots T_n\rangle$, and the pair parameters-body, respectively, of the function f in C, if any, looked for in C first, then in its parent contract. Finally, the subtyping relation $C \leq D$ is the reflexive and transitive closure of the inheritance relation. Subtyping is extended to function types as usual: $\langle T, T_1 \dots T_n\rangle \leq \langle T', T'_1 \dots T'_n\rangle$ if $T'_i \leq T_i$, for $i \in 1..n$, and $T \leq T'$.

Reduction rules (CTX) and (CTX-REVERT) are straightforward. In rule (ACCESS) the semantics is as expected, with an additional check that the state variable s exists in both contracts C and D and that the type obtained at runtime is a subtype of that statically computed from the cast type. Otherwise, a `revert[rte]` is

$e ::= \ldots \mid \iota_D^C$ runtime expr. $\quad v ::= \iota_D^C \mid \mathtt{u} \mid \mathsf{n} \mid a$ value

$c ::= \langle e, \beta\rangle$ configuration $\quad \mathcal{E} ::= [\,] \mid \mathcal{E}.s \mid \mathcal{E}\,.\,s\texttt{=}e' \mid \iota\,.\,s\texttt{=}\mathcal{E} \mid \{\,T\,x\texttt{=}\mathcal{E}\,\texttt{;}\,e\} \ldots$ evaluation context

$$\textsc{(ctx)}\ \frac{\langle e,\beta\rangle \longrightarrow \langle e',\beta'\rangle}{\langle \mathcal{E}[e],\beta\rangle \longrightarrow \langle \mathcal{E}[e'],\beta'\rangle} \qquad \textsc{(ctx-revert)}\ \frac{}{\langle \mathcal{E}[\mathtt{revert}[\lambda]],\beta\rangle \longrightarrow \langle \mathtt{revert}[\lambda],\beta\rangle}$$

$$\textsc{(access)}\ \frac{}{\langle \iota_D^C.s,\beta\rangle \longrightarrow \langle \beta(\iota_D).i,\beta\rangle}\ \begin{array}{l}\mathsf{svar}(D,s)=i\\ \mathsf{svartype}(D,s) \le \mathsf{svartype}(C,s)\end{array}$$

$$\textsc{(access-rte)}\ \frac{}{\langle \iota_D^C.s,\beta\rangle \longrightarrow \langle \mathtt{revert}[\mathtt{rte}],\beta\rangle}\ \mathsf{svartype}(D,s) \not\le \mathsf{svartype}(C,s)$$

$$\textsc{(assign)}\ \frac{}{\langle \iota_D^C\,.\,s\texttt{=}v,\beta\rangle \longrightarrow \langle v,\beta[\iota_D.i\texttt{=}v]\rangle}\ \begin{array}{l}\mathsf{svar}(D,s)=i\\ \mathsf{svartype}(C,s) \le \mathsf{svartype}(D,s)\end{array}$$

$$\textsc{(assign-rte)}\ \frac{}{\langle \iota_D^C\,.\,s\texttt{=}v,\beta\rangle \longrightarrow \langle \mathtt{revert}[\mathtt{rte}],\beta\rangle}\ \mathsf{svartype}(C,s) \not\le \mathsf{svartype}(D,s)$$

$$\textsc{(dec)}\ \frac{}{\langle \{\,T\,x\texttt{=}v\,\texttt{;}\,e\},\beta\rangle \longrightarrow \langle e[v/x],\beta\rangle} \qquad \textsc{(cast)}\ \frac{}{\langle C(a),\beta\rangle \longrightarrow \langle \iota_D^C,\beta\rangle}\ \langle \iota_D, a\rangle \in \mathsf{dom}(\beta)$$

$$\textsc{(get-addr)}\ \frac{}{\langle \texttt{address(}\iota\texttt{)},\beta\rangle \longrightarrow \langle a,\beta\rangle}\ \langle \iota, a\rangle \in \mathsf{dom}(\beta) \qquad \textsc{(get-bal)}\ \frac{}{\langle \texttt{balance(}a\texttt{)},\beta\rangle \longrightarrow \langle \mathsf{n},\beta\rangle}\ \beta(a).\mathtt{v} = \mathsf{n}$$

$$\textsc{(new)}\ \frac{}{\langle \texttt{new } C\texttt{.value(}\mathsf{n}\texttt{).sender(}a^{\mathsf{s}}\texttt{)(}vs\texttt{)},\beta\rangle \longrightarrow \langle \iota_C,\beta[\langle\iota_C,a\rangle \leftarrow vs][a^{\mathsf{s}}.\mathtt{v} \overset{\mathsf{n}}{\rightsquigarrow} a.\mathtt{v}]\rangle}\ \begin{array}{l}|vs| = |\mathsf{svars}(C)|\\ \langle\iota_C,a\rangle \notin \mathsf{dom}(\beta)\\ \beta(a^{\mathsf{s}}).\mathtt{v} \geqslant \mathsf{n}\end{array}$$

$$\textsc{(new-neg)}\ \frac{}{\langle \texttt{new } C\texttt{.value(}\mathsf{n}\texttt{).sender(}a^{\mathsf{s}}\texttt{)(}vs\texttt{)},\beta\rangle \longrightarrow \langle \mathtt{revert}[\mathtt{neg}],\beta\rangle}\ \beta(a^{\mathsf{s}}).\mathtt{v} < \mathsf{n}$$

$$\textsc{(invk)}\ \frac{}{\begin{array}{l}\langle \iota_D^C\,.\,f\,\texttt{.value(}\mathsf{n}\texttt{).sender(}a^{\mathsf{s}}\texttt{)(}vs\texttt{)},\beta\rangle \longrightarrow\\ \qquad\qquad \langle e[\iota_D/\mathtt{this}][a^{\mathsf{s}}/\mathtt{msg.s}][\mathsf{n}/\mathtt{msg.v}][vs/xs],\beta[a^{\mathsf{s}}.\mathtt{v} \overset{\mathsf{n}}{\rightsquigarrow} a.\mathtt{v}]\rangle\end{array}}\ \begin{array}{l}\mathsf{fbody}(D,f) = \langle xs, e\rangle\\ \mathsf{ftype}(D,f) \le \mathsf{ftype}(C,f)\\ \langle \iota_D, a\rangle \in \mathsf{dom}(\beta)\\ \beta(a^{\mathsf{s}}).\mathtt{v} \geqslant \mathsf{n}\end{array}$$

$$\textsc{(invk-rte)}\ \frac{}{\langle \iota_D^C\,.\,f\,\texttt{.value(}\mathsf{n}\texttt{).sender(}a^{\mathsf{s}}\texttt{)(}vs\texttt{)},\beta\rangle \longrightarrow \langle \mathtt{revert}[\mathtt{rte}],\beta\rangle}\ \mathsf{ftype}(D,f) \not\le \mathsf{ftype}(C,f)$$

$$\textsc{(invk-neg)}\ \frac{}{\langle \iota_D^C\,.\,f\,\texttt{.value(}\mathsf{n}\texttt{).sender(}a^{\mathsf{s}}\texttt{)(}vs\texttt{)},\beta\rangle \longrightarrow \langle \mathtt{revert}[\mathtt{neg}],\beta\rangle}\ \beta(a^{\mathsf{s}}).\mathtt{v} < \mathsf{n}$$

$$\textsc{(transf)}\ \frac{}{\begin{array}{l}\langle a\,\texttt{.transfer(}\mathsf{n}\texttt{).sender(}a^{\mathsf{s}}\texttt{)},\beta\rangle \longrightarrow\\ \qquad\qquad \langle e[\iota_C/\mathtt{this}][a^{\mathsf{s}}/\mathtt{msg.sender}][\mathsf{n}/\mathtt{msg.value}],\beta[a^{\mathsf{s}}.\mathtt{v} \overset{\mathsf{n}}{\rightsquigarrow} a.\mathtt{v}]\rangle\end{array}}\ \begin{array}{l}\langle \iota_C, a\rangle \in \mathsf{dom}(\beta)\\ \mathsf{fbody}(C,\mathtt{fb}) = \langle \epsilon, e\rangle\\ \beta(a^{\mathsf{s}}).\mathtt{v} \geqslant \mathsf{n}\end{array}$$

$$\textsc{(transf-neg)}\ \frac{}{\langle a\,\texttt{.transfer(}\mathsf{n}\texttt{).sender(}a^{\mathsf{s}}\texttt{)},\beta\rangle \longrightarrow \langle \mathtt{revert}[\mathtt{neg}],\beta\rangle}\ \beta(a^{\mathsf{s}}).\mathtt{v} < \mathsf{n}$$

$$\textsc{(transf-fb)}\ \frac{}{\langle a\,\texttt{.transfer(}\mathsf{n}\texttt{).sender(}a^{\mathsf{s}}\texttt{)},\beta\rangle \longrightarrow \langle \mathtt{revert}[\mathtt{rte}],\beta\rangle}\ \begin{array}{l}\langle \iota_C, a\rangle \in \mathsf{dom}(\beta)\\ \mathsf{fbody}(C,\mathtt{fb}) \text{ undefined}\end{array}$$

Fig. 4. FS: operational semantics

raised, see rule (ACCESS-RTE), whose side condition is intended to cover also the case where s is not defined in both contracts. A symmetric check is performed in rules (ASSIGN) and (ASSIGN-RTE). Indeed, as mentioned above, in Solidity no runtime checks are performed in a cast, but they are postponed when the reference is actually used.[4] This is modeled in rule (CAST), where an address is converted to the corresponding reference. The runtime effect is just to tag the reference with the static type for future usage checks; in particular no subtyping constraint, like $D \leq C$, is enforced.

In rule (DEC), local variable declarations have the standard substitution semantics. Rules (GET-ADDR) and (GET-BAL) are straightforward. In rule (NEW), a fresh instance of C is added in the blockchain, with state and balance initialized to the tuple of values and amount provided as arguments of the constructor invocation. Furthermore, the balance of the contract instance at address a^{s} is decremented of the same amount, provided that this would not make the balance negative, otherwise a `revert[neg]` is raised, see rule (NEW-NEG).

In rule (INVK), the parameters and body of the function f defined in the contract of the receiver are retrieved from the contract table, through the auxiliary function fbody. Analogously to rules (ACCESS) and (ASSIGN), a check is performed that the function f exists in both contracts C and D and the function type obtained at runtime is a subtype of that statically computed from the cast type, otherwise a `revert[rte]` is raised, see rule (INVK-RTE). The invocation is reduced to the function body where `this` and formal parameters have been replaced by the receiver and the arguments vs, as in standard FJ, and, moreover, `msg.sender` and `msg.value` have been replaced by address a^{s} and amount n, respectively. Finally, the balance of the contract instance at address a^{s} is decremented of the same amount, provided that this would not make the balance negative, otherwise a `revert[neg]` is raised, see rule (INVK-NEG).

While functions are invoked on contract references, the `transfer` construct is used with addresses. In rule (TRANSFER), an amount of n Wei is transferred from the balance of the contract instance at address a^{s} to that at a, provided that this would not make the sender balance negative, otherwise, a `revert[neg]` is raised, see rule (TRANSFER-NEG). Moreover, the fallback function is implicitly invoked, if any, otherwise a `revert[rte]` is raised, see rule (TRANSFER-FB).

4 Type system

The typing judgment has shape $\Gamma;\mathcal{I};\mathcal{A} \vdash e : T$, where Γ is a finite map from variables to types, $\mathcal{I}$ and $\mathcal{A}$ are sets of references and addresses, respectively. As for the reduction relation, it is implicitly parameterized by a contract table.

Typing rules are given in Fig.5; they are mostly straightforward. Note that rule (T-REF) assigns the static type of a contract reference by looking at its superscript. According to the semantics of cast, rule (T-CAST) just checks that the

[4] The Solidity semantics is actually more involved, since in some cases an attempt is made to convert values from the provided to the expected type. No documentation about the precise behavior is available.

$$\textsc{(t-var)}\ \frac{}{\Gamma;\mathcal{I};\mathcal{A} \vdash x : T}\ \Gamma(x) = T \qquad \textsc{(t-unit)}\ \frac{}{\Gamma;\mathcal{I};\mathcal{A} \vdash \mathtt{u} : \mathtt{unit}} \qquad \textsc{(t-nat)}\ \frac{}{\Gamma;\mathcal{I};\mathcal{A} \vdash \mathtt{n} : \mathtt{uint}}$$

$$\textsc{(t-addr)}\ \frac{}{\Gamma;\mathcal{I};\mathcal{A} \vdash a : \mathtt{address}}\ a \in \mathcal{A} \qquad \textsc{(t-ref)}\ \frac{}{\Gamma;\mathcal{I};\mathcal{A} \vdash \iota_D^C : C}\ \iota_D \in \mathcal{I}$$

$$\textsc{(t-access)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : C}{\Gamma;\mathcal{I};\mathcal{A} \vdash e.s : T}\ \mathsf{svartype}(C,s) = T \qquad \textsc{(t-cast)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : \mathtt{address}}{\Gamma;\mathcal{I};\mathcal{A} \vdash C(e) : C}$$

$$\textsc{(t-assign)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : C \qquad \Gamma;\mathcal{I};\mathcal{A} \vdash e' : T'}{\Gamma;\mathcal{I};\mathcal{A} \vdash e.s\mathtt{=}e' : T}\ \begin{array}{l}\mathsf{svartype}(C,s) = T\\ T' \leq T\end{array}$$

$$\textsc{(t-dec)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : T \qquad \Gamma, x : T;\mathcal{I};\mathcal{A} \vdash e' : T'}{\Gamma;\mathcal{I};\mathcal{A} \vdash \{T\, x\mathtt{=}e\mathtt{;}\, e'\} : T'} \qquad \textsc{(t-revert)}\ \frac{}{\Gamma;\mathcal{I};\mathcal{A} \vdash \mathtt{revert}[\lambda] : T}$$

$$\textsc{(t-get-addr)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : C}{\Gamma;\mathcal{I};\mathcal{A} \vdash \mathtt{address(}e\mathtt{)} : \mathtt{address}} \qquad \textsc{(t-get-bal)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : \mathtt{address}}{\Gamma;\mathcal{I};\mathcal{A} \vdash \mathtt{balance(}e\mathtt{)} : \mathtt{uint}}$$

$$\textsc{(t-new)}\ \frac{\begin{array}{l}\Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{v}} : \mathtt{uint}\\ \Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{s}} : \mathtt{address} \qquad \Gamma;\mathcal{I};\mathcal{A} \vdash e_i : T_i' \ \forall i \in 1..n\end{array}}{\Gamma;\mathcal{I};\mathcal{A} \vdash \mathtt{new}\ C\mathtt{.value(}e^{\mathsf{v}}\mathtt{).sender(}e^{\mathsf{s}}\mathtt{)(}e_1,\ldots,e_n\mathtt{)} : C}\ \begin{array}{l}\mathsf{svars}(C) = T_1\, s_1 \ldots T_n\, s_n\\ T_i' \leq T_i\ \forall i \in 1..n\end{array}$$

$$\textsc{(t-invk)}\ \frac{\begin{array}{l}\Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{v}} : \mathtt{uint} \qquad \Gamma;\mathcal{I};\mathcal{A} \vdash e : C\\ \Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{s}} : \mathtt{address} \qquad \Gamma;\mathcal{I};\mathcal{A} \vdash e_i : T_i' \ \forall i \in 1..n\end{array}}{\Gamma;\mathcal{I};\mathcal{A} \vdash e.f\mathtt{.value(}e^{\mathsf{v}}\mathtt{).sender(}e^{\mathsf{s}}\mathtt{)(}e_1,\ldots,e_n\mathtt{)} : T}\ \begin{array}{l}\mathsf{ftype}(C,f) = \langle T,\, T_1 \ldots T_n\rangle\\ T_i' \leq T_i\ \forall i \in 1..n\end{array}$$

$$\textsc{(t-transfer)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : \mathtt{address} \qquad \Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{v}} : \mathtt{uint} \qquad \Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{s}} : \mathtt{address}}{\Gamma;\mathcal{I};\mathcal{A} \vdash e\mathtt{.transfer(}e^{\mathsf{v}}\mathtt{).sender(}e^{\mathsf{s}}\mathtt{)} : \mathtt{unit}}$$

$$\textsc{(t-conf)}\ \frac{\emptyset;\mathcal{I};\mathcal{A} \vdash e : T \qquad \mathcal{I};\mathcal{A} \vdash \beta}{\mathcal{I};\mathcal{A} \vdash \langle e, \beta\rangle : T}$$

Fig. 5. FS: typing rules for expressions and configurations

expression to be cast has type `address` without performing any additional type check. The typing judgment is extended to configurations in rule (t-conf), requiring that both the expression to be evaluated and the blockchain are well-typed according to the same sets of addresses and contract references. Moreover, the expression should contain no free variables. The judgment $\mathcal{I};\mathcal{A} \vdash \beta$ holds if:

$$a \in \mathcal{A} \text{ iff } a \in \mathsf{dom}(\beta),\ \ \iota \in \mathcal{I} \text{ iff } \iota \in \mathsf{dom}(\beta), \text{ and } \beta(\iota_C) = \langle v_1, ..., v_n, \mathsf{n}\rangle \text{ implies}$$
$$\mathsf{svars}(C) = T_1\, s_1 ... T_n\, s_n \text{ and, for all } i \in 1..n,\ \emptyset;\mathcal{I};\mathcal{A} \vdash v_i : T_i' \text{ with } T_i' \leq T_i.$$

Finally, the judgment $\mathcal{I};\mathcal{A} \vdash CT$ means that the contract table is *well-formed* w.r.t. existing contract references and addresses. We omit the complete formal definition, reported in [7], since it is essentially as in FJ. Informally, it ensures that all used contract names are declared, the inheritance relation is acyclic, there is no state variable hiding, no function overloading and safe function overriding. Moreover, each function definition should be well-typed in the following sense: if $\mathsf{ftype}(C,f) = \langle T,\, T_1 \ldots T_n\rangle$ and $\mathsf{fbody}(C,f) = \langle x_1 ... x_n, e\rangle$ then $\mathtt{this}{:}C,\ \mathtt{msg.sender}{:}\mathtt{address},\ \mathtt{msg.value}{:}\mathtt{uint},\ x_1{:}T_1, .., x_n{:}T_n\ ;\emptyset\ ;\mathcal{A} \vdash e : T'$ with $T' \leq T$. Notice that the previous judgment assumes an empty set of references since the code of contract functions can refer to contract instances and EOAs only by means of (public) addresses.

We write $\longrightarrow^\star$ for the reflexive and transitive closure of $\longrightarrow$ and $c \not\longrightarrow$ if there is no c' s.t. $c \longrightarrow c'$. In the theorem we implicitly assume that the underlying class table is well-formed w.r.t. $\mathcal{I}$ and $\mathcal{A}$.

Theorem 1 (Soundness). *If $\mathcal{I}; \mathcal{A} \vdash c : T$, $c \longrightarrow^\star c'$, and $c' \not\longrightarrow$, where $c' = \langle e', \beta' \rangle$, then either e' is a value or $e' = \texttt{revert}[\lambda]$ for some λ.*

This soundness theorem states that the Solidity type system prevents stuck execution, but not runtime type errors. This is quite dangerous and can lead to Ether indefinitely locked into a contract or to unexpected runtime `revert`s.

For instance, consider a blockchain storing at address a_B an instance of the `Bank` contract in Fig.3, and at address a_D an instance of a contract D that does not define a fallback function. Assume that the contract at a_D successfully deposited 100 Wei in the bank, and now wants to withdraw part of them. The function call `Bank(`a_B`).withdraw.value(`0`).sender(`a_D`)(50)` successfully compiles, but reduces to a_D`.transfer(50)` that raises a `revert`[`rte`] exception since a_D refers to a contract that does not define a valid fallback fallback function. Therefore, the withdrawal transaction aborts, causing loss of gas fee in the real Ethereum scenario. Moreover, since deployed contract code cannot be updated, the money already deposited by a_D in the bank a_B is indefinitely locked.

The whole problem lies in the way the `address` type is handled: neither Solidity nor the EVM provides additional information on the contract stored at that address. Solidity addresses represent an untyped way to access contract instances, much as `void *` pointers in C. Such pointers allow extreme flexibility, but they are really difficult to deal with, since programmers have to know what they are doing and how to do so, in order to avoid subtle bugs.

5 Refined type system

This section refines the type system of FS in order to more safely access contract instances through their address. Indeed, the resulting type system enjoys a more powerful soundness property, that is, well-typed programs never reduce to a `revert`[`rte`] exception. The key idea is to enrich the `address` type so to type information about the contracts the addresses refer to. That is, `address`$\langle C \rangle$ is the type of the addresses of instances of the contract C. This richer type is mostly useful when typing the implicit `msg.sender` variable, used in function bodies to refer to the address of the caller. Indeed, well-typed expressions such as C`(msg.sender).`f`.value(n)()` or `msg.sender.transfer(n)` reduce to a `revert`[`rte`] exception if `msg.sender` is bound to the address of a contract that has not type C or has no fallback function. On the other hand, by enriching the contract functions' signatures with the address type of the implicit sender parameter, we can let the compiler check the safety of callbacks expressions similar to the ones above, that are notoriously vulnerable Solidity programming patterns.

$fd ::= T\ f$ `<`S`>(`$T_1\,x_1, \ldots, T_n\,x_n$`){ return` e`; }` function declaration
$T ::= C \mid$ `unit` $\mid$ `uint` $\mid$ `address`$\langle C\rangle$ type

$$\text{(T-ADDR)}\ \frac{}{\Gamma;\mathcal{I};\mathcal{A} \vdash a : \texttt{address}\langle C\rangle}\ \mathcal{A}(a) = C$$

$$\text{(T-GET-ADDR)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : C}{\Gamma;\mathcal{I};\mathcal{A} \vdash \texttt{address(}e\texttt{)} : \texttt{address}\langle C\rangle} \qquad \text{(T-CAST)}\ \frac{\Gamma;\mathcal{I};\mathcal{A} \vdash e : \texttt{address}\langle D\rangle}{\Gamma;\mathcal{I};\mathcal{A} \vdash C(e) : C}\ D \leq C$$

$$\text{(T-NEW)}\ \frac{\begin{array}{l}\Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{v}} : \texttt{uint}\\ \Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{s}} : \texttt{address}\langle S\rangle \quad \Gamma;\mathcal{I};\mathcal{A} \vdash e_i : T_i' \ \forall i \in 1..n\end{array}}{\Gamma;\mathcal{I};\mathcal{A} \vdash \texttt{new}\ C\texttt{.value(}e^{\mathsf{v}}\texttt{).sender(}e^{\mathsf{s}}\texttt{)(}e_1, \ldots, e_n\texttt{)} : C}\ \begin{array}{l}\mathsf{svars}(C) = T_1\,s_1 \ldots T_n\,s_n\\ T_i' \leq T_i\ \forall i \in 1..n\end{array}$$

$$\text{(T-INVK)}\ \frac{\begin{array}{ll}\Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{v}} : \texttt{uint} & \Gamma;\mathcal{I};\mathcal{A} \vdash e : C\\ \Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{s}} : \texttt{address}\langle S'\rangle & \Gamma;\mathcal{I};\mathcal{A} \vdash e_i : T_i' \ \forall i{\in}1..n\end{array}}{\Gamma;\mathcal{I};\mathcal{A} \vdash e.f\texttt{.value(}e^{\mathsf{v}}\texttt{).sender(}e^{\mathsf{s}}\texttt{)(}e_1, \ldots, e_n\texttt{)} : T}\ \begin{array}{l}\mathsf{ftype}(C,f){=}\langle T, S, T_1 \ldots T_n\rangle\\ S' \leq S\\ T_i' \leq T_i\ \forall i \in 1..n\end{array}$$

$$\text{(T-TRANSFER)}\ \frac{\begin{array}{l}\Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{v}} : \texttt{uint}\\ \Gamma;\mathcal{I};\mathcal{A} \vdash e^{\mathsf{s}} : \texttt{address}\langle S'\rangle \quad \Gamma;\mathcal{I};\mathcal{A} \vdash e : \texttt{address}\langle C\rangle\end{array}}{\Gamma;\mathcal{I};\mathcal{A} \vdash e\texttt{.transfer(}e^{\mathsf{v}}\texttt{).sender(}e^{\mathsf{s}}\texttt{)} : \texttt{unit}}\ \begin{array}{l}\mathsf{ftype}(C,\texttt{fb}){=}\langle\texttt{unit}, S, \epsilon\rangle\\ S' \leq S\end{array}$$

Fig. 6. FS$^+$: changes to syntax and typing rules

Formally, the refined calculus, called FS$^+$, is obtained by applying the changes in Fig.6 to the syntax of FS. In function declarations, the metavariable S (for "sender") ranges over contract names, and the meaning is that the function f can be called only by contracts or EOAs whose address has (a subtype of) type `address`$\langle S\rangle$. The subtyping relation is extended to address types in covariant way, that is, `address`$\langle C\rangle \leq$ `address`$\langle D\rangle$ holds if $C \leq D$.

The typing rules of FS$^+$ are obtained by applying the changes in Fig.6. Moreover, in the typing judgement, $\mathcal{A}$ is no longer a set, but a map from addresses to contract names. The judgment $\mathcal{I};\mathcal{A} \vdash \beta$ must additionally require that if $\langle \iota_C, a\rangle \in \mathsf{dom}(\beta)$ then $\mathcal{A}(a) = C$. Finally, function types become triples, $\langle T, S, T_1 \ldots T_n\rangle \leq \langle T', S', T_1' \ldots T_n'\rangle$ additionally requires $S' \leq S$, and the requirement on well-formedness of function bodies becomes the following: if $\mathsf{ftype}(C,f) = \langle T, S, T_1 \ldots T_n\rangle$ and $\mathsf{fbody}(C,f){=}\langle x_1 ... x_n, e\rangle$ then
$\texttt{this}{:}C, \texttt{msg.sender}{:}\texttt{address}\langle S\rangle, \texttt{msg.value}{:}\texttt{uint}, x_1{:}T_1, .., x_n{:}T_n; \emptyset; \mathcal{A} \vdash e : T'$
with $T' \leq T$. The refined rule (T-CAST) now statically checks that the expression to be cast evaluates to the address of an instance of contract D which is a subtype of the target of the cast. In rules (T-INVK) and (T-TRANSFER), the additional side condition requires the type of the sender e^{s} to be a subtype of the type S of the expected caller of the function f and `fb`, respectively, as specified in their refined signature.

The type system of FS$^+$ enjoys a stronger soundness property: `revert`[`rte`] errors are statically prevented, so the only possible runtime errors remain those due to a negative account balance. In other terms, cast expressions or money transfers that would lead to unsafe usage of contract members or calls to an undefined fallback function are now ruled out at compile-time.

Theorem 2 (Soundness). *If $\mathcal{I};\mathcal{A} \vdash c : T$, $c \longrightarrow^{\star} c'$, and $c' \not\rightarrow$, where $c' = \langle e', \beta' \rangle$, then either e' is a value or $e' = \texttt{revert}[\texttt{neg}]$.*

By taking advantage of this more powerful typing, the `Bank` contract in Fig.3 can be refined into the following safer smart contract:

```
contract Bank {
  mapping(address<Top_fb> => uint) amounts;
  unit deposit <Top_fb>() {...}
  unit withdraw  <Top_fb>(uint n) {...}
}
```

Function bodies do not change and are hence omitted. We assume a contract $\texttt{Top}_{\texttt{fb}}$ which only contains a `fb` function with empty body and `Top` sender parameter. Address types used in the mapping to index the banks' clients refer to such contract name. This type is also used in the refined signature of the two contract functions, so to (statically) ensure that their caller contract actually provides a fallback function. Therefore, coming back to the example discussed in Section 4, if $a_B : \texttt{address}\langle\texttt{Bank}\rangle$ and $a_D : \texttt{address}\langle D\rangle$ where the contract D has no fallback function, the function call $\texttt{Bank}(a_B)\texttt{.withdraw.value(0).sender}(a_D)\texttt{(50)}$ does not compile anymore, since the new rule (T-INVK) requires $D \leq \texttt{Top}_{\texttt{fb}}$, which is not true. The runtime error occurring when trying to tranfer money to a_D is then statically prevented. Similarly, the contract stored at a_D cannot even call the deposit function, thus preventing also the deposit of money that cannot be withdrawn anymore.

The introduction of the type $\texttt{address}\langle C\rangle$ and the corresponding typing rules, are of course incompatible with the legacy Solidity code, that would not be accepted anymore by the new compiler. Nonetheless, a direct default mapping is easily definable by mapping each occurrence of the type `address` to $\texttt{address}\langle\texttt{Top}\rangle$ and by refining each function signature so to use `Top` as supertype of the function's sender. We shall also provide a flag (`--notopcast`) in the new compiler to disable the refined rule (T-CAST) when $D = \texttt{Top}$and use the standard rule (T-CAST) of Section 4. Indeed, the refined rule would rule out any cast having $\texttt{address}\langle\texttt{Top}\rangle$ as actual type of e, since for all type C, $\texttt{Top} \not\leq C$.Cleary, by using such a default mapping, no additional guarantees can be statically checked on the contracts code, however, retro-compatibility with the Solidity smart contracts already deployed on the blockchain, whose code cannot be updated anymore, is guaranteed.

To take advantage of the full power of the refined typing, the major effort required to Solidity programmers is to annotate each function with the required (super)type of the caller. We then put forward a couple of new convenient annotations, in line with the Solidity programming style, that provides a number of modifiers to annotate functions, e.g., the `payable` marker in Fig.1. Since it is often the case that type constraints refer to contracts that provide (at least) a fallback function, the keyword `payableaddress` can be introduced as a syntactic sugar for the type `address`$\langle$`Top`$_{\texttt{fb}}\rangle$, and the function marker `payback` can be used to indicate that the function potentially sends Ether back to its caller. Therefore, the Solidity `Bank` contract given in Fig.1 could be simply rewritten into the following code, where function bodies are as in Fig.1:

```
contract Bank {
   mapping (payableaddress => uint) private amounts;
   function deposit() payable payback {...}
   function withdraw() payback {...}
}
```

Instead, to enforce type-safe callbacks in functions code, programmers are required to explicitly express the type constraint they require on contracts callers. However, this requirement actually supports a safer programming discipline.

6 Conclusions

We developed semantic foundations of smart contract programming, by formalizing the core of the Solidity language and type system. The FS calculus allows one to precisely define the behavior of smart contract programs, thus it represents a fundamental step to develop automatic program analysis tools. The FS's type system clarifies the type soundness of the Solidity compiler, pointing out its limitations. We then put forward a refined type discipline that statically captures a larger class of errors, such as unsafe casts, unsafe callbacks and money transfers that cannot be accepted by contracts because they lack the fallback function. We discussed how such extension impacts on the Solidity legacy code so to actually provide a safer programming discipline that is retrocompatibile with smart contracts already deployed on the blockchain. Finally, the FS calculus highlights the connection between objects and smart contracts, thus opening the way to reuse the type theory of OOLs in the context of Solidity, and dually to adapt the refined typing of FS$^+$ to the case of distributed objects.

Related work A number of proposals have been developed to improve the security and correctness of Ethereum smart contracts. A stream of works, e.g.,[9, 4, 8], addresses the problem at the bytecode level: the semantics of EVM bytecode is formalized and smart contracts properties are verified by means of static analysis tools operating on the corresponding bytecode. Among the ones addressing the problem at the programming language level, Zeus [11] translates Solidity code into LLVM bytecode [12], leveraging abstract interpretation and symbolic model checking analysis techniques. SmartCheck [14], instead, attempts to detect vulnerabilities representing Solidity code as an XML tree, and then running XPath queries on it. Contracts code is fully covered, but the use of XPath leads to a higher rate of false positives. However, these tools are based on limited formal foundations of the language they operate on, and they come into play when a contract is fully defined. We rather think that by enhancing the Solidity compiler's ability to statically rule out harmful code, we support a safer programming discipline, where programmers can write smart contracts that are (more) correct by construction. The work presented in [3] operates in this direction, and provides a preliminary compiler extension encoding Solidity code into SMT formulas to check simple properties, such as the division by zero. Similarly, the tool developed in [13] encodes a subset of Solidity into SMT formulas and uses symbolic model checking to verify some properties about smart contracts behaviour, including temporal ones.

The first attempt to formalize Solidity is presented in [5]. This work develops a language-based approach for smart contracts verification using F* [1]. In brief, a small subset of Solidity is translated into F*, whose type system is afterwards used to detect vulnerable patterns, such as reentrancy. Even though the results are encouraging, the subset of Solidity is too small (neither transfer or cast expressions are considered), and an external language, F*, is used.

To the best of our knowledge, this paper, together with its preliminary version [7], is the first work aiming at directly formalizing the semantics and the type soundness of the Solidity source code, so to enhance the use of its compiler as a convenient building tool.

References

1. F* programming language. https://www.fstar-lang.org/.
2. Solidity documentation. https://solidity.readthedocs.io/en/develop/index.html. Release 0.4.25.
3. L. Alt and C. Reitwießner. Smt-based verification of solidity smart contracts. In *Leveraging Applications of Formal Methods, Verification and Validation. Industrial Practice ISoLA*, pages 376–388, 2018.
4. S. Amani, M. Bégel, M. Bortin, and M. Staples. Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In *Certified Programs and Proofs*, pages 66–77. ACM, 2018.
5. K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Gollamudi, G. Gonthier, N. Kobeissi, N. Kulatova, A. Rastogi, T. Sibut-Pinote, N. Swamy, et al. Formal verification of smart contracts: Short paper. In *ACM Workshop on Programming Languages and Analysis for Security*, pages 91–96. ACM, 2016.
6. V. Buterin. A next-generation smart contract and decentralized application platform (white paper). Technical report, 2014.
7. M. Di Pirro. How solid is Solidity? An in-dept study of Solidity's type safety. Master's thesis, Università di Padova, Sept. 2018. http://tesi.cab.unipd.it/61297/.
8. I. Grishchenko, M. Maffei, and C. Schneidewind. A semantic framework for the security analysis of Ethereum smart contracts. In *Principles of Security and Trust*, pages 243–269. Springer, 2018.
9. E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. M. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu. KEVM: A complete formal semantics of the ethereum virtual machine. In *Computer Security Foundations Symposium, CSF*, pages 204–217, 2018.
10. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
11. S. Kalra, S. Goel, M. Dhawan, and S. Sharma. Zeus: Analyzing safety of smart contracts. Network and Distributed System Security Symposium, 2018.

12. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE, 2004.
13. E. Shishkin. Debugging smart contract's business logic using symbolic model-checking. *arXiv preprint arXiv:1812.00619*, 2018.
14. S. Tikhomirov, E. Voskresenskaya, I. Ivanitskiy, R. Takhaviev, E. Marchenko, and Y. Alexandrov. Smartcheck: Static analysis of Ethereum smart contracts. 2018.
15. G. Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum project yellow paper*, 151:1–32, 2014.