SSCalc
A Calculus for Solidity Smart Contracts

Diego Marmsoler[1] [0000−0003−2859−7673] and Billy Thornton[1]

University of Exeter, Exeter, UK
{d.marmsoler, b.thornton}@exeter.ac.uk

Abstract. Smart contracts are programs stored on the blockchain, often developed in a high-level programming language, the most popular of which is Solidity. Smart contracts are used to automate financial transactions and thus bugs can lead to large financial losses. With this paper, we address this problem by describing a verification environment for Solidity in Isabelle/HOL. To this end, we first describe a calculus to reason about Solidity smart contracts. The calculus is formalized in Isabelle/HOL and its soundness is mechanically verified. Then, we describe a Verification Condition Generator to automate the use of the calculus. Our approach can be used to verify the functional correctness of Solidity smart contracts. To demonstrate this, we use it to verify a simple token implemented in Solidity. Our results show that the framework has the potential to significantly reduce the verification effort compared to verifying directly from the semantics.

Keywords: Smart Contracts · Solidity · Program Verification · Isabelle/Solidity.

1 Introduction

Blockchain [33] is a novel technology for storing data in a decentralized manner, providing *transparency*, *security*, and *trust*. Although the technology was originally invented to enable cryptocurrencies, it quickly found applications in several other domains, such as *finance* [24], *healthcare* [5], *land management* [12], and even *identity management* [43]. According to McKinsey, blockchain had a market capitalization of more than $150B in 2018 [8] and Gartner predicts its business value to be $3.1T by 2030 [19].

One important innovation that comes with blockchains are so-called *smart contracts*. These are digital contracts that are automatically executed once certain conditions are met and that are used to automate transactions on the blockchain. For instance, a payment for an item might be released instantly once the buyer and seller have met all specified parameters for a deal. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions [42].

Technically, a smart contract is code that is deployed to a blockchain and that can be executed by sending special transactions to it. Smart contracts are usually developed in a high-level programming language, the most popular of which is *Solidity* [18]. Solidity is based on the Ethereum Virtual Machine (EVM) and thus it works on all EVM-based smart contract platforms, such as Ethereum, Avalanche, Moonbeam, Polygon,

This version of the article has been accepted for publication in the proceedings of the 21st International Conference on Software Engineering and Formal Methods (SEFM), after peer review and is subject to Springer Nature's AM terms of use, but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online.

BSC, and more. As of today, 85% of all smart contracts are developed using Solidity [25] and according to a 2019 survey, Solidity is by far the most popular language used by blockchain developers (in fact it outranked the second most popular language by 100%) [38].

As for every computer program, *smart contracts may contain bugs that can be exploited*. However, since smart contracts are often used to automate financial transactions, such exploits may result in huge economic losses. For example, in 2016 a vulnerability in an Ethereum smart contract was exploited resulting in a loss of approximately $60M [6]. More recently, hackers exploited a vulnerability in the DeFi-platform Poly Network to steal $600M [34]. As another example, an incorrectly initialized contract was the root cause of the Parity Wallet bug that froze $280M [36]. In general, it is estimated that since 2019, more than $5B was stolen due to vulnerabilities in smart contracts [13].

The high impact of vulnerabilities in smart contracts together with the fact that once deployed to the blockchain, they cannot be updated or removed easily, makes it important to "get them right" before they are deployed. As a result, there has been a growing amount of work to verify smart contracts (see [2] for an overview). Most of the existing work focuses on the automatic detection of certain types of vulnerabilities, such as *re-entrancy*, *integer overflow/underflow*, or *call-stack depth limit*. However, they do not allow for the verification of general *functional correctness*.

Thus, in the following paper, we present SSCalc, a framework for the verification of the functional correctness of Solidity smart contracts. To this end, the contributions of this paper are twofold. First, we describe a calculus to reason about Solidity smart contracts. Our calculus extends traditional calculi, used to reason about sequential and object-oriented programs [3], with new rules to capture the characteristics of smart contracts. We formalized the calculus in Isabelle/HOL [35] and verified its soundness mechanically from the formal semantics of Solidity developed in previous work [26]. Second, we developed a verification condition generator (VCG) to automate the use of the calculus. The VCG is implemented in Isabelle/Eisbach [30] and consists of a set of proof methods, which can be used to verify contract invariants and pre-/postconditions for (internal) methods.

To evaluate our approach, we verified a basic implementation of a token [39] in Solidity with and without using the calculus. Our results show that the calculus has the potential to significantly reduce the effort required to verify a Solidity smart contract. Without the calculus, verification required ca. 3250 lines of Isabelle/Isar code whereas using the calculus reduced it to ca. 700 lines.

2 Background

Our calculus is based on the denotational semantics of a subset of Solidity described in [26,27,28]. Our subset supports the following features of Solidity:

- *Fixed-size integer types* of various lengths with *support for overflow* and corresponding arithmetic.
- Domain-specific primitives to *transfer funds* and *query balances*.
- Different types of stores, such as *storage*, *memory*, *calldata*, and *stack*.

- Complex data types, such as *hash-maps* and *arrays*.
- *Assignment with different semantics*, depending on the location of the involved data types (deep vs. shallow copy of complex data types).
- An abstract *gas model* that can be instantiated with concrete gas costs for each statement.
- *Internal* and *external method declarations* and the ability to *transfer funds* with external method calls.
- Declaration of *fallback methods* which are implicitly executed with monetary transfers.

2.1 Inductive Data Types

Our semantics is formalized in higher-order logic using inductive data types [9]. To this end, we use **bold** font for types and Roman font for type constructors.

For a datatype

$$\textbf{nat} \stackrel{\text{def}}{=} \mathrm{Zero}() \mid \mathrm{Suc}(\textbf{nat})$$

we shall often use the case construct to match a variable against constructors:

$$\mathrm{dec}(x) \stackrel{\text{def}}{=} \text{case } x \text{ of } \begin{cases} \mathrm{Zero}() & \Rightarrow \mathrm{Zero}() \\ \mathrm{Suc}(n) & \Rightarrow n \end{cases}$$

We shall also use

$$\textbf{type}_\bot \stackrel{\text{def}}{=} \bot \cup \{x_\bot \mid x \in \textbf{type}\}$$

to denote the type that adds a *distinct* element $\bot$ to the elements of **type**.

2.2 State Monad

Our semantics is defined using the concept of a state monad [14,40]. To this end we first define a result type as follows:

$$\mathbf{result}(n, e) \stackrel{\text{def}}{=} \mathrm{N}(n) \mid \mathrm{E}(e)$$

The type $\mathbf{result}$ is defined over two type parameters, n, and e, which denote the type for normal and erroneous return values, respectively.

We can then define a state monad as follows:

$$\mathbf{sm}(a, e, s) \stackrel{\text{def}}{=} s \to \mathbf{result}(a \times s, e)$$

The monad requires three type parameters: type a for return values, type e for exceptions, and type s for states. Such a monad either updates state s and returns an element of type a or returns an exception of type e.

2.3 State

In Solidity users and contracts are identified by *addresses* with associated balances. Moreover, a contract operates over different types of stores: a *stack* and *memory* to keep volatile data, as well as *storage* to keep persistent data. Finally, in Solidity computation consumes so-called *gas*. Thus, a state is defined as follows:

$$\mathbf{state} \stackrel{\text{def}}{=} \mathbf{accounts} \times \mathbf{stack} \times \mathbf{memory} \times (\mathbf{address} \to \mathbf{storage}) \times \mathbf{nat}$$

where $\mathbf{accounts}$ map addresses to balances and $\mathbf{nat}$ represents the available gas. Data types $\mathbf{stack}$, $\mathbf{memory}$, and $\mathbf{storage}$ represent the different types of stores and map locations to values (note also that each address has its private storage). In the following, we use $\mathrm{acc}(st)$, $\mathrm{sck}(st)$, $\mathrm{mem}(st)$, $\mathrm{sto}(st)$, and $\mathrm{gas}(st)$ to access the account, stack, memory, storage, and gas components of state st. Moreover, we shall use

$$st(\!|\mathrm{gas} := g, \mathrm{acc} := a, \mathrm{sck} := k, \mathrm{mem} := m, \mathrm{sto} := s|\!)$$

to update the gas, account, stack, memory, and storage of state st to g, a, k, m, and s, respectively.

2.4 Exceptions

In the following, we distinguish between two types of exceptions to signal erroneous executions. Thus, we define the following type for exceptions:

$$\mathbf{error} \stackrel{\text{def}}{=} \mathrm{Gas}() \mid \mathrm{Err}()$$

An exception Gas occurs whenever a computation runs out of gas. All other erroneous situations are captured by exception Err.

3 Calculus

In the following, we describe a *weakest precondition* calculus [17] to reason about Solidity. To this end, we fix the following four parameters:

- ep: A *procedure environment* assigning contracts to their addresses.
- ad: The *address* of the contract to be verified
- $\mathrm{contract}$: The *implementation* of *methods* of the contract to be verified.
- fb: The *implementation* of the *fallback* method of the contract to be verified.

In addition, we assume that the procedure environment ep associates the address ad of the contract to be verified with its implementation $\mathrm{contract}$ and fb:

$$\mathrm{ep}(\mathrm{ad}) = (\mathrm{contract}, \mathrm{fb})_\bot$$

We can then define the weakest precondition for our state monad as follows:

$$\mathrm{wp} : \mathbf{sm}((), \mathbf{error}, \mathbf{state}) \times (\mathbf{state} \to \mathbf{b}) \times (\mathbf{error} \to \mathbf{b}) \to \mathbf{state} \to \mathbf{b}$$

$$\mathrm{wp}(f, P, E) \stackrel{\text{def}}{=} \lambda st.\ \mathrm{case}\ f(st)\ \mathrm{of} \begin{cases} \mathrm{N}(\_, st') & \Rightarrow P(st') \\ \mathrm{E}(e) & \Rightarrow E(e) \end{cases}$$

where $()$ denotes the unit type (the type with only one element $()$) and $\mathbf{b}$ is the boolean type. It defines the weakest precondition of statement f, state predicate P, and exception predicate E. If f, executed in state st, terminates successfully with state st', the weakest precondition equals P evaluated over st'. On the other hand, if the statement throws an exception e, the weakest precondition equals E evaluated over e.

A user usually prefers to specify correctness criteria using Hoare triples instead of weakest preconditions. Thus, we further introduce the validity of a Hoare triple for a statement. To this end, we first specify the notion of a state predicate and an exception predicate:

$$\mathbf{spred} \stackrel{\text{def}}{=} \mathbf{accounts} \times \mathbf{stack} \times \mathbf{memory} \times (\mathbf{address} \to \mathbf{storage}) \to \mathbf{b}$$
$$\mathbf{epred} \stackrel{\text{def}}{=} \mathbf{error} \to \mathbf{b}$$

Now we can define validity as follows:

$$\{\_\}\ \_\ \{\_\}\{\_\} : \mathbf{spred} \times \mathbf{sm}((), \mathbf{error}, \mathbf{state}) \times \mathbf{spred} \times \mathbf{epred} \to \mathbf{b}$$
$$\{P\}\ f\ \{Q\}\{E\} \stackrel{\text{def}}{=} \forall st.\ P(\mathrm{acc}(st), \mathrm{sck}(st), \mathrm{mem}(st), \mathrm{sto}(st))$$
$$\implies \text{case } f(st) \text{ of } \begin{cases} \mathrm{N}(\_, st') & \Rightarrow \begin{array}{l}\mathrm{gas}(st') \leq \mathrm{gas}(st) \wedge \\ Q(\mathrm{acc}(st'), \mathrm{sck}(st'), \mathrm{mem}(st'), \mathrm{sto}(st'))\end{array} \\ \mathrm{E}(e) & \Rightarrow E(e) \end{cases}$$

A Hoare triple $\{P\}\ f\ \{Q\}\{E\}$ is valid if for every state st that satisfies the state predicate P, statement f either terminates in a state st' that satisfies state predicate Q or leads to an error e that satisfies error predicate E. Note that we also require that execution does not increase the amount of available gas.

To validate our definitions, we proved the following lemma about the relationship between the validity of Hoare triples and weakest preconditions:

Lemma 1.

$$\{P\}\ f\ \{Q\}\{E\} \iff \begin{array}{l}\forall s.\ P(\mathrm{acc}(s), \mathrm{sck}(s), \mathrm{mem}(s), \mathrm{sto}(s)) \\ \quad \implies \mathrm{wp}(f, (\lambda s.\ Q(\mathrm{acc}(s), \mathrm{sck}(s), \mathrm{mem}(s), \mathrm{sto}(s))), E, s)\end{array}$$

3.1 Basic Rules

Our calculus includes rules for all the basic statements: WP_SKIP for the empty statement, WP_ASSIGN for assignments, WP_COMP for compositions, WP_ITE for conditionals, and WP_WHILE for while loops (which require the specification of an invariant). These rules are mostly standard; and thus, they are not discussed further here.

There are, however, two particularities worth mentioning. First, each rule needs to deal with the case that there might not be enough gas available to execute a statement. Second, assignments are somewhat special in Solidity because the semantics of assignments depend on the location of the expression on the left and right. Each side may evaluate to a location on either stack, calldata, memory, or storage. Thus, when verifying an assignment in Solidity, we must consider 16 different cases and two additional error cases.

3.2 Method Invocation

In Solidity, a contract may have two types of methods. *Internal* methods can only be called internally from within the same contract. *External* methods, on the other hand, can only be called by other contracts. In addition, a Solidity contract has a designated *fallback* method. This method is invoked whenever the contract receives some payments or if a method is called that does not exist.

The following rule allows us to verify (recursive) method calls:

$$\textsc{Wp\_External\_Invoke\_Transfer}\quad \frac{\forall st'.\ \text{gas}\ st' \leq \text{gas}\ st \implies P(i,p,p',p_f,p'_f,st') \vdash Q(i,p,p',p_f,p'_f,st)}{P(i,p,p',p_f,p'_f,st')}$$

where predicates P and Q are defined below and $A \vdash C$ denotes that C is derivable from A in our calculus.

The rule requires the specification of several parameters:

- i: An invariant for the contract's private storage and balance.
- p, p': Pre/postconditions for each internal method.
- p_f, p'_f: One pre/postcondition for the contract's fallback method.

We can then use the rule to establish a predicate P for an arbitrary state st' by proving Q for an arbitrary state st. While proving Q, the rule allows us to assume P for all states st' with less (or equal) gas than st.

In the following, we are going to discuss predicates P and Q in more detail.

Predicate P. This predicate is defined as follows:

$$P(i,p,p',p_f,p'_f,st) \stackrel{\text{def}}{=} P_{\text{e}}(i,st) \wedge P_{\text{i}}(p,p',st) \wedge P_{\text{fi}}(p_f,p'_f,st) \wedge P_{\text{fe}}(i,st)$$

It *establishes the weakest precondition* for method *calls* and *transfer* statements.

$P_{\text{e}}(iv,st)$. This predicate establishes the weakest precondition for *external method calls*. In Solidity, external method calls can be used to invoke methods of other contracts deployed to the blockchain. Moreover, it is possible to transfer funds from the caller to the callee with each call. In the following, we use $\text{External}(ad',i,xe,val)$ to denote an external method call where

- ad' is an expression denoting the address of the called contract.
- i is the identifier of the method to be called.
- xe is a list containing actual parameters for the method.
- val is an expression denoting the amount of funds sent with the call.

Predicate $P_{\text{e}}(iv,st)$ can now be defined as in Fig. 1, where $\text{address}(ev)$ denotes the address associated with an environment ev, $\text{expr}(ex,ev,cd,st,g)$ evaluates an expression ex using an environment ev and calldata cd in a state st, ng is the updated gas value $\text{gas}(st) - \text{costs}(\text{External}(ad',i,xe,val),ev,cd,st)$, $\lceil x \rceil$ converts a string to an integer, and E denotes the exception predicate $\lambda e.\ e = \text{Gas} \ \vee\ e = \text{Err}$.

$$\forall ev, ad', i, xe, val, cd.$$
$$\text{address}(ev) = \text{ad} \wedge \tag{1}$$
$$(\forall adv, g, v, t, g'.\ adv \neq \text{ad} \ \wedge \ adv \in \text{dom}(\text{ep}) \wedge$$
$$\text{expr}(ad', ev, cd, st(\!|\text{gas} := \text{ng}|\!), \text{ng}) = \text{N}((\text{V}(adv), \text{V}(\text{TAddr})), g) \wedge$$
$$\text{expr}(val, ev, cd, st(\!|\text{gas} := g|\!), g) = \text{N}((\text{V}(v), \text{V}(t)), g')$$
$$\implies iv(\text{sto}(st)(\text{ad}), \lceil \text{acc}(st)(\text{ad}) \rceil \text{ - } \lceil v \rceil)) \tag{2}$$
$$\implies \text{wp}(\text{stmt}(\text{External}(ad', i, xe, val), ev, cd),$$
$$\lambda st.\ iv(\text{sto}(st)(\text{ad}), \lceil \text{acc}(st)(\text{ad}) \rceil), \text{E}, st) \tag{3}$$

Fig. 1. Definition of $P_{\text{e}}(iv, st)$.

Eq. 3 establishes the weakest precondition of invariant iv and error predicate E for an external method call executed in state st. Eq. 1 ensures that the address of the currently executing contract is indeed address ad of the contract to be verified (fixed at the beginning of Sect. 3). Eq. 2 requires that the invariant holds before executing the call. However, note that we require the invariant to hold on a modified version of the balance. In particular, value v (which is obtained by evaluating expression val) is deduced from the actual balance of the contract. This is because the actual call transfers v funds from the caller to the callee. Thus, to ensure that the invariant holds after the call, we must ensure that the invariant holds on a balance in which the value is already deduced.

$P_{\text{i}}(pre, post, st)$. This predicate establishes the weakest precondition for *internal method calls*. In Solidity, internal method calls can only invoke internal methods of the currently executing contract. In the following, we use $\text{Invoke}(i, xe)$ to denote a call to an internal method i with actual parameters xe. $P_{\text{i}}(pre, post, st)$ can now be defined as in Fig. 2,

$$\forall ev, i, xe, cd.$$
$$\text{address}(ev) = \text{ad} \wedge \tag{4}$$
$$(\forall fp, e_l, cd_l, k_l, m_l, g.$$
$$\text{load}(\text{False}, fp, xe, \text{nev}, \emptyset, \emptyset, \text{mem}(st), ev, cd, st(\!|\text{gas} := \text{ng}|\!), \text{ng}) =$$
$$\text{N}((e_l, cd_l, k_l, m_l), g)$$
$$\implies pre(\lceil \text{acc}(st)(\text{ad}) \rceil, \text{sto}(st)(\text{ad}), e_l, cd_l, k_l, m_l)) \tag{5}$$
$$\implies \text{wp}(\text{stmt}(\text{Invoke}(i, xe), ev, cd),$$
$$\lambda st.\ post(i)(\lceil \text{acc}(st)(\text{ad}) \rceil, \text{sto}(st)(\text{ad})), \text{E}, st) \tag{6}$$

Fig. 2. Definition of $P_{\text{i}}(pre, post, st)$.

where $\mathrm{load}(cp, fp, xe, nev, cd', sck', mem', ev, cd, st)$ initializes formal parameters fp with actual parameters xe, nev is a fresh environment for the execution of the method body, and ng is the updated gas value $\mathrm{gas}(st) - \mathrm{costs}(\mathrm{Invoke}(i, xe), ev, cd, st)$.

Eq. 6 establishes the weakest precondition of the method's postcondition $post(i)$ and error predicate E for an internal method call executed in state st. Again, Eq. 4 ensures that the currently executing contract is the one to be verified (with address ad). Eq. 5, however, requires that the method's precondition holds before the execution of the call. Note that the precondition is a predicate over 6 parameters: the current contracts balance and private store, as well as the environment created by loading the actual parameters (environment e_l, calldata cd_l, stack k_l, and memory m_l).

$P_{\mathrm{fe}}(iv, st)$. This predicate establishes the weakest precondition for external transfers. In Solidity, transfer statements can be used to transfer funds from contracts to accounts. In the following, we use $\mathrm{Transfer}(ad', ex)$ to denote a transfer statement in which

- ad' is an expression denoting the address of the receiver, and
- ex is an expression denoting the amount to be transferred.

Note that, if the receiving address belongs to a contract, a transfer implicitly triggers the execution of a so-called fallback method. Thus, $P_{\mathrm{fe}}(iv, st)$ can be defined as in Fig. 3, where ng is the updated gas value $\mathrm{gas}(st) - \mathrm{costs}(\mathrm{Transfer}(ad', ex), ev, cd, st)$.

$$
\begin{aligned}
&\forall ev, ex, ad', cd.\\
&\quad \mathrm{address}(ev) = \mathrm{ad} \wedge && (7)\\
&\quad \big(\forall adv, g.\ \mathrm{expr}(ad', ev, cd, st(\!|\mathrm{gas} := \mathrm{ng}|\!), \mathrm{ng}) = \mathrm{N}((\mathrm{V}(adv), \mathrm{V}(\mathrm{TAddr})), g)\\
&\qquad \Longrightarrow\ adv \neq \mathrm{ad}\big) \wedge && (8)\\
&\quad \big(\forall adv, g, v, t, g'.\ adv \neq \mathrm{ad} \wedge\\
&\qquad \mathrm{expr}(ad', ev, cd, st(\!|\mathrm{gas} := \mathrm{ng}|\!), \mathrm{ng}) = \mathrm{N}((\mathrm{V}(adv), \mathrm{V}(\mathrm{TAddr})), g) \wedge\\
&\qquad \mathrm{expr}(ex, ev, cd, st(\!|\mathrm{gas} := g|\!), g) = \mathrm{N}((\mathrm{V}(v), \mathrm{V}(t)), g')\\
&\qquad \Longrightarrow\ iv(\mathrm{sto}(st)(\mathrm{ad}), \lceil \mathrm{acc}(st)(\mathrm{ad}) \rceil \text{ - } \lceil v \rceil\ )\big) && (9)\\
&\Longrightarrow\ \mathrm{wp}(\mathrm{stmt}(\mathrm{Transfer}(ad', ex), ev, cd),\\
&\qquad \lambda st.\ iv(\mathrm{sto}(st)(\mathrm{ad}), \lceil \mathrm{acc}(st)(\mathrm{ad}) \rceil), \mathrm{E}, st) && (10)
\end{aligned}
$$

Fig. 3. Definition of $P_{\mathrm{fe}}(iv, st)$.

Eq. 10 establishes the weakest precondition of invariant iv and error predicate E for a transfer statement executed in state st. Again, Eq. 7 ensures that the currently executing contract is the one we want to verify (on address ad). In addition, Eq. 8 requires that the receiving contract is different from the executing contract (because for self-transfers we have a different rule). Finally, Eq. 9 requires the invariant to hold before the transfer statement is executed. Again, we require that the invariant holds

on a balance in which the value is already deduced from the balance of the currently executing contract.

$P_{\text{fi}}(pre_f, post_f, st)$. This predicate establishes the weakest precondition for internal transfers. The rule is similar to P_{fe}, but since control is not passed on to an external contract we may use pre-/post-conditions instead of an invariant. Thus, the definition of P_{fi} is the same as that of P_{fe} (shown in Fig. 3) with the following changes:

- Eq. 8 is changed to $adv = \text{ad}$.
- In Eq. 9 $iv(\dots)$ is replaced with $pre_f(\text{sto}(st)(\text{ad}), \lceil \text{acc}(st)(\text{ad}) \rceil)$.
- In Eq. 10 $iv(\dots)$ is replaced with $post_f(\text{sto}(st)(\text{ad}), \lceil \text{acc}(st)(\text{ad}) \rceil)$.

Note that we require pre_f to hold for the original balance $\text{acc}(st)(\text{ad})$ and *not* for the modified version as in Eq. 9. This is because an internal transfer does not modify the current contract's balance, because the amount is first deduced from it but then added again.

Predicate Q. This predicate is defined as follows:

$$Q(i, p, p', p_f, p'_f, st) \stackrel{\text{def}}{=} Q_{\text{e}}(i, st) \wedge Q_{\text{i}}(p, p', st) \wedge Q_{\text{fi}}(p_f, p'_f, st) \wedge Q_{\text{fe}}(p_f, p'_f, st)$$

It denotes *proof obligations* for different types of methods.

$Q_e(iv, st)$. This predicate denotes proof obligations for external methods; that is, it tells us what we need to verify to establish the weakest precondition of an invariant for an external method. It is defined in Fig. 4 where ng is the updated gas value $\text{gas}(st') - \text{costs}(\text{External}(adex, mid, xe, val), ev, cd, st')$, nev is a fresh environment for the execution of the method body, and $\text{transfer}(s, r, v, a)$ is used to transfer funds of value v from sending address s to receiving address r for accounts a.

Eq. 15 shows the actual statement we need to verify, i.e., that the weakest precondition of invariant iv and error predicate E for method body f holds in state st' with gas g'', accounts acc, stack k_l, and memory m_l. The statement needs to be verified only for external methods invoked from a context outside the contract to be verified. Thus, Eq. 11 requires that f is indeed the body of an external method mid of the contract to be verified (contract) and Eq. 12 ensures that the method is invoked from outside (i.e. an address different from the contract to be verified).

To verify Eq. 15 we can assume that the invariant holds for the state in which f will be executed (Eq. 14). This is because we know that the invariant holds when control leaves the current contract. Thus, if another contract is to call back into the current contract the invariant must still hold. Note, however, that the invariant holds only on a modified balance for contract ad. This is because the calling contract may send some funds v with the method call which are then transferred to the receiving contract ad. Thus, since we know that the invariant holds before transferring the funds, we need to deduce v from the balance of ad after the transfer.

When verifying Eq. 15 we can also assume that the current level of gas is less than or equal to the original amount of gas (Eq. 13). This is an important property because it allows us to use all P predicates from WP_EXTERNAL_INVOKE_TRANSFER, which, according to the rule, can only be assumed for states with less or equal gas than the original state.

$$
\begin{aligned}
&\forall mid, fp, f, ev.\\
&\quad \mathrm{contract}(mid) = \mathrm{Method}(fp, \mathrm{True}, f)_\perp \wedge && (11)\\
&\quad \mathrm{address}(ev) \neq \mathrm{ad} && (12)\\
&\implies (\forall adex, cd, st', xe, val, g, v, t, g', e_l, cd_l, k_l, m_l, g'', acc.\\
&\qquad \mathrm{expr}(adex, ev, cd, st'(\!|\mathrm{gas} := \mathrm{ng}|\!), \mathrm{ng}) = \mathrm{N}((\mathrm{V}(\mathrm{ad}), \mathrm{V}(\mathrm{TAddr})), g) \wedge\\
&\qquad \mathrm{expr}(val, ev, cd, st'(\!|\mathrm{gas} := g|\!), g) = \mathrm{N}((\mathrm{V}(v), \mathrm{V}(t)), g') \wedge\\
&\qquad \mathrm{load}(\mathrm{True}, fp, xe, \mathrm{nev}, \emptyset, \emptyset, \emptyset, ev, cd, st'(\!|\mathrm{gas} := g'|\!), g') =\\
&\qquad\quad \mathrm{N}((e_l, cd_l, k_l, m_l), g'')\\
&\qquad g'' \leq \mathrm{gas}(st) \wedge && (13)\\
&\qquad \mathrm{transfer}(\mathrm{address}(ev), \mathrm{ad}, v, \mathrm{acc}(st'(\!|\mathrm{gas} := g''|\!))) = acc_\perp \wedge\\
&\qquad iv(\mathrm{sto}(st')(\mathrm{ad}), \lceil acc(\mathrm{ad}) \rceil - \lceil v \rceil) && (14)\\
&\quad \implies \mathrm{wp}(\mathrm{stmt}(f, e_l, cd_l), \lambda st.\ iv(\mathrm{sto}(st)(\mathrm{ad}), \lceil \mathrm{acc}(st)(\mathrm{ad}) \rceil), \mathrm{E},\\
&\qquad\quad st'(\!|\mathrm{gas} := g'', \mathrm{acc} := acc, \mathrm{sck} := k_l, \mathrm{mem} := m_l|\!))) && (15)
\end{aligned}
$$

Fig. 4. Definition of $Q_e(iv, st)$.

$Q_i(pre, post, st)$. This predicate denotes proof obligations for internal methods, i.e., it tells us what we need to verify to establish the weakest precondition of a method's postcondition from its precondition. In Fig. 5 ng is the updated gas value $\mathrm{gas}(st') - \mathrm{costs}(\mathrm{Invoke}(i, xe), ev, cd, st')$ and nev is a fresh environment for the execution of the method body.

Eq. 20 states what we need to verify, i.e., that the weakest precondition of the post-condition $post(mid)$ associated with method mid and error predicate E for method body f holds in state st' with gas g, stack k_l, and memory m_l. The statement needs to be verified only for internal methods invoked from a context inside the contract to be verified. Thus, Eq. 16 requires that f is indeed the body of an internal method mid of the contract to be verified (contract) and Eq. 17 ensures that the method is invoked from inside (i.e., from address ad).

Again, when verifying Eq. 20, we can assume that the available gas is less or equal to the original amount of gas (Eq. 18). Moreover, we can also assume that the methods precondition holds for the environment in which method body f will be executed (Eq. 19). The statement needs to be verified only for external methods invoked from a context outside the contract to be verified. Thus, Eq. 11 requires that f is indeed the body of an external method mid of the contract to be verified (contract) and Eq. 12 ensures that the method is invoked from outside (i.e., an address different from the contract to be verified).

$Q_{\mathrm{fe}}(iv, st)$. This predicate denotes proof obligations to establish the weakest precondition of an invariant for fallback methods executed as a result of an external transfer.

$$
\begin{aligned}
&\forall mid, fp, f, ev.\\
&\quad \mathit{contract}(mid) = \mathrm{Method}(fp, \mathrm{False}, f)_{\perp} \wedge && (16)\\
&\quad \mathrm{address}(ev) = \mathrm{ad} && (17)\\
&\implies (\forall cd, st', i, xe, e_l, cd_l, k_l, m_l, g.\\
&\qquad \mathrm{load}(\mathrm{False}, fp, xe, \mathrm{nev}, \emptyset, \emptyset, \mathrm{mem}(st'), ev, cd, st'(\!|\mathrm{gas} := \mathrm{ng}|\!), \mathrm{ng}) =\\
&\qquad\quad \mathrm{N}((e_l, cd_l, k_l, m_l), g) \wedge\\
&\qquad g \leq \mathrm{gas}(st) \wedge && (18)\\
&\qquad \mathit{pre}(mid)(\lceil \mathrm{acc}(st')(\mathrm{ad}) \rceil, \mathrm{sto}(st')(\mathrm{ad}), e_l, cd_l, k_l, m_l) && (19)\\
&\qquad \implies \mathrm{wp}(\mathrm{stmt}(f, e_l, cd_l), \lambda st.\ \mathit{post}(mid)(\lceil \mathrm{acc}(st)(\mathrm{ad}) \rceil, \mathrm{sto}(st)(\mathrm{ad})), \mathrm{E},\\
&\qquad\quad st'(\!|\mathrm{gas} := g, \mathrm{sck} := k_l, \mathrm{mem} := m_l|\!)) && (20)
\end{aligned}
$$

Fig. 5. Definition of $Q_i(pre, post, st)$.

It is defined in Fig. 6 where ng is the updated gas value $\mathrm{gas}(st') - c$ and nev is a fresh environment for the execution of the fallback method.

$$
\begin{aligned}
&\forall ev.\ \mathrm{address}(ev) \neq \mathrm{ad} && (21)\\
&\implies (\forall ex, cd, st', adex, v, t, g, g', acc, c.\\
&\qquad \mathrm{expr}(adex, ev, cd, st'(\!|\mathrm{gas} := \mathrm{ng}|\!), \mathrm{ng}) = \mathrm{N}((\mathrm{V}(\mathrm{ad}), \mathrm{V}(\mathrm{TAddr})), g) \wedge\\
&\qquad \mathrm{expr}(ex, ev, cd, st'(\!|\mathrm{gas} := g|\!), g) = \mathrm{N}((\mathrm{V}(v), \mathrm{V}(t)), g') \wedge\\
&\qquad g' \leq \mathrm{gas}(st) \wedge && (22)\\
&\qquad \mathrm{transfer}(\mathrm{address}(ev), \mathrm{ad}, v, \mathrm{acc}(st')) = acc_{\perp} \wedge\\
&\qquad iv(\mathrm{sto}(st')(\mathrm{ad}), \lceil acc(\mathrm{ad}) \rceil \text{ - } \lceil v \rceil) && (23)\\
&\qquad \implies \mathrm{wp}(\mathrm{stmt}(\mathrm{fb}, \mathrm{nev}, \emptyset), \lambda st.\ iv(\mathrm{sto}(st)(\mathrm{ad}), \lceil \mathrm{acc}(st)(\mathrm{ad}) \rceil), E,\\
&\qquad\quad st'(\!|\mathrm{gas} := g', \mathrm{sck} := \emptyset, \mathrm{acc} := acc, \mathrm{mem} := \emptyset|\!)) && (24)
\end{aligned}
$$

Fig. 6. Definition of $Q_{\mathrm{fe}}(iv, st)$.

Eq. 24 states what we need to verify, i.e., that the weakest precondition of the invariant iv and error predicate E for our contracts fallback method fb holds in state st' with gas g', a fresh stack and memory, and account acc. Since it only needs to be verified for external transfers, Eq. 21 ensures that the transfer statement is issued externally.

Again, when verifying Eq. 24, we can assume that the current level of gas is less than or equal to the original level (Eq. 22). Moreover, we know that the invariant holds when the transfer occurs. Thus, since the transfer adds v funds to the balance of contract ad, we can assume that the invariant holds when we deduce v again.

$Q_{\text{fi}}(pre_f, post_f, st)$. This predicate denotes proof obligations for internal transfers, i.e., it tells us what we need to verify to establish the weakest precondition of the post-condition of the fallback method from its precondition. Its definition is similar to that of Q_{fe}, with modifications similar to those required for P_{fi} above.

4 Formalization in Isabelle/HOL

The complete calculus is formalized in Isabelle/HOL, and its soundness is mechanically verified[1] from our semantics.

4.1 Verification of Soundness

The verification of soundness of our rules is mostly standard, except for rule WP_EXTERNAL_INVOKE_TRANSFER. In particular, external method calls and transfer statements transfer control to another contract. Thus, we must ensure that other contracts can never change the validity of an invariant. To this end, we prove the following lemma:

$$\begin{aligned}
&\forall st'.\ \text{address}(ev) \neq \text{ad} \wedge && (25)\\
&\quad iv(\text{sto}(st)(\text{ad}), \lceil \text{acc}(st)(\text{ad}) \rceil) \wedge && (26)\\
&\quad \text{stmt}(f, ev, cd, st) = \text{N}((), st') \wedge && (27)\\
&\quad \forall st'.\ \text{gas}(st') < \text{gas}(st) \implies Q_e(iv, st') \wedge && (28)\\
&\quad \forall st'.\ \text{gas}(st') < \text{gas}(st) \implies Q_{fe}(iv, st') && (29)\\
&\quad \implies iv(\text{sto}(st')(\text{ad}), \lceil \text{acc}(st')(\text{ad}) \rceil) && (30)
\end{aligned}$$

With this lemma we verified that an invariant iv for the storage and balance of contract ad is preserved (Eq. 26 and Eq. 30) by the execution of arbitrary statements f (Eq. 27) executed in a different context from that of ad (Eq. 25), given that the external methods (Eq. 28) and the fallback method (Eq. 29) of contract ad preserve the invariant. Eq. 28 and Eq. 29 are particularly important here because f may contain statements that call back to ad and thus execute code that may potentially impact iv.

Since our semantics is formalized as a deep embedding in Isabelle/HOL, the statement above can be easily proven by structural induction on f.

4.2 Automation

To support users in applying the calculus for the verification of Solidity smart contracts we implemented a verification condition generator (VCG). The VCG automates the use of the calculus and leaves the user with a so-called verification condition that needs to be discharged to ensure the correctness of the contract. The VCG is implemented in Isabelle/Eisbach [30] and consists of different methods to support the verification of different types of statements[2].

[1] Theory `Weakest_Precondition.thy` from the accompanying artefact [29].
[2] Section "Verification Condition Generator" in `Weakest_Precondition.thy` [29].

5 Methodology

In the following section, we demonstrate our approach using a simple example. To this end, consider the contract depicted in Listing 1.1, which stores an unsigned integer `x` (and possibly other variables not shown). Moreover, it provides an internal method `int1`, which calls an external method `ext()` of a contract with address `ad1` and sends 1 ether with it. It also provides another internal method `int2`, which calls `int1`. In addition, it provides an external method `ext`, which transfers 1 ether to a contract with address `ad2` and another 1 ether to itself. Finally, it also has a fallback method which does not have a name.

```
1  contract Example {
2    uint x;
3    ...
4    function int1(uint y, ...) internal {
5      ...
6      ad1.call.value(1 ether)(abi.encodeWithSignature("ext()"));
7      ...
8    }
9    function int2(int y, ....) internal {
10     ...
11     int1(5, ...);
12     ...
13   }
14   function ext() external {
15     ...
16     ad2.transfer(1 ether);
17     ...
18     address(this).transfer(1 ether);
19     ...
20   }
21   function () external payable {
22     ...
23   }
24 }
```

Listing 1.1. A simple example contract.

To verify the contract using our calculus, we first need to specify the following:

- An *invariant*: A predicate over the contract's member variables (including, for example, x) and the contract's balance.
- *Preconditions* for internal methods `int1` and `int2`: Predicates over the method's formal parameters (including, for example, y), the contract's member variables (including, for example, x), and the contract's balance.
- *Postconditions* for internal methods `int1` and `int2`: Predicates over the contract's member variables and the contract's balance.
- A *precondition* and *postcondition* for the contract's fallback method: A predicate over the contract's member variables and the contract's balance.

We then need to verify the following:

- Executing the body of `int1` (Ln. 5 - Ln. 7) in a state in which its precondition holds, leads to a state in which its postcondition holds.
- Executing the body of `int2` (Ln. 10 - Ln. 12) in a state in which its precondition holds, leads to a state in which its postcondition holds.
- Executing the body of `ext` (Ln. 15 - Ln. 19) in a state in which the invariant holds, leads to a state in which the invariant holds again.
- Executing the body of the fallback method (Ln. 22) in a state in which its precondition holds, leads to a state in which its postcondition holds, and executing the body in a state in which the invariant holds, leads to a state in which the invariant holds again.

To verify the above proof obligations we can use the rules of the calculus and the following assumptions:

- If the invariant holds before executing Ln. 6, then it holds also after executing it.
- If the precondition associated with `int1` holds before the execution of Ln. 11, then the corresponding postcondition holds after executing Ln. 11.
- If the invariant holds before executing Ln. 16, then it holds also after executing it.
- If the fallback methods precondition holds before the execution of Ln. 18, then its postcondition holds after executing Ln. 18.

6 Case Study: Verified Banking

In the following, we use our calculus to verify a contract that implements a simple banking system.

6.1 The Contract

The contract should allow users to deposit funds and later withdraw them. A possible implementation is provided by the contract shown in Listing 1.2.

```
1  contract Bank {
2    mapping(address => uint256) balances;
3
4    function deposit() external payable {
5      balances[msg.sender] = balances[msg.sender] + msg.value;
6    }
7
8    function withdraw() external {
9      uint256 bal = balances[msg.sender];
10     balances[msg.sender] = 0;
11     msg.sender.transfer(bal);
12   }
13 }
```

Listing 1.2. A simple banking contract.

The contract has one member variable `balances` to keep track of all the balances. Moreover, it provides two methods to deposit and withdraw funds. When a contract calls `deposit` with some funds, the funds are transferred to the Bank contract and the amount is kept in `msg.value`. Thus, method `deposit` simply adds the value to the balance of the calling contract to keep track of how much each contract contributed to the funds of the banking contract. A contract can call `withdraw` to get its funds back. To this end, the banking contract first sets the caller's internal balance to 0 (Ln. 10) and then returns the corresponding funds (Ln. 11). Note that it is important to *first* update the internal balance before transferring the money. Thus, the contract is secure against so-called re-entrancy attacks [4]. However, the question remains whether the contract is indeed functionally correct or if it is exposed to other vulnerabilities.

6.2 Formalizing the Contract

To answer this question, we first need to formalize the contract in our semantics. To this end, we need to provide definitions for the parameters of our calculus described at the beginning of Sect. 3:

$$\text{contract} = \begin{cases} \text{“}balances\text{”} & \mapsto & \text{Var}(\text{STMap}(\text{TAddr}, \text{STValue}(\text{TUInt}(256)))) \\ \text{“}deposit\text{”} & \mapsto & \text{Method}([], \text{True}, \text{deposit}) \\ \text{“}withdraw\text{”} & \mapsto & \text{Method}([], \text{True}, \text{withdraw}) \end{cases}$$

$$\text{fb} = \text{Skip}$$

The contract is formalized as a mapping from identifiers to corresponding members. While “$balances$” refers to a variable, “$deposit$” and “$withdraw$” refer to *external* methods with body deposit and withdraw defined as in Listing 1.2. The contract does not define a fallback method; thus fb is defined as Skip.

6.3 Specification of Properties

The property we want to verify for our contract is that the relationship between the sum of all stored balances and the internal balance of the contract is preserved through the execution of each external method.

Thus, we first formalize the following invariant:

$$\begin{aligned} \text{iv}(bal, s, a) &\stackrel{\text{def}}{=} a - \text{sum}(s) \geq bal \wedge bal \geq 0 \wedge \text{pos}(s) \\ \text{sum}(s) &\stackrel{\text{def}}{=} \sum_{\{(ad,x) \mid s(ad+\text{“.”}+\text{“}balances\text{”})=x_\perp\}} \lceil x \rceil \\ \text{pos}(s) &\stackrel{\text{def}}{=} \forall ad, x.\ s(ad + \text{“.”} + \text{“}balances\text{”}) = x_\perp \implies \lceil x \rceil \geq 0 \end{aligned}$$

The important part here is the first conjunction in the definition of iv: $a - \text{sum}(s) \geq bal$. Here, a represents the funds available to our banking contract and $\text{sum}(s)$ represents the sum of all its stored balances. Thus, the formula requires that the difference between these two balances is bound by a certain value bal.

Now, we can formalize the properties thst we want to verify using the Hoare triple notation introduced in Sect. 3:

$$\{I\}\ \text{stmt}(\text{External}(\text{Address}(\text{ad}), \text{“}deposit\text{”}, [], val), env, cd)\ \{I\}\{E\}$$
$$\{I\}\ \text{stmt}(\text{External}(\text{Address}(\text{ad}), \text{“}withdraw\text{”}, [], val), env, cd)\ \{I\}\{E\}$$

where $I((a, \_, \_, s)) \stackrel{\text{def}}{=} iv(bal, s(ad), \lceil a(ad) \rceil)$, $E(e) \stackrel{\text{def}}{=} e = \text{Gas} \vee e = \text{Err}$, and $\text{address}(env) \neq \text{ad}$.

6.4 Verification

As discussed in Sect. 3.2, Solidity implicitly triggers the execution of a so-called fallback method whenever money is transferred to a contract. In particular, if another contract calls `withdraw`, the transfer statement in Ln. 11 of Listing 1.2 triggers the execution of the callee's fallback method. Thus, as we do not know all potential contracts that call `withdraw`, we need to verify the invariant for all possible implementations.

To evaluate our approach, we verified the above property twice: from its semantics without using the calculus [28], and using our calculus [29]. Without the calculus, verifying the above property required ca. 3 250 lines of Isabelle/Isar code. Using the calculus reduced it to ca. 700 lines.

7 Related Work

Since Solidity is the most popular language for developing smart contracts there has been growing interest in formalizing its semantics. Bhargavan et al. [10], for example, provide a semantics of Solidity in F*. Crosara et al. [16] describe an operational semantics for a subset of Solidity. Hajdu and Jovanovic [21], provide a formalization of Solidity in terms of a simple SMT-based intermediate language. In addition, Zakrzewski [44] describes a big-step semantics of a small subset of Solidity and Yang and Lei [41] describe a formalization of a subset of Solidity in Coq [37]. Moreover, Jiao et al. [22,23], provide a formalization of Solidity in $\mathbb{K}$. Finally, Cassez et al. [11] provide an implementation of Solidity in Dafny. All of these works provide important contributions towards a better understanding of Solidity. The focus of our work was to provide a framework for the verification of smart contracts written in Solidity and while it is possible to verify them directly from the semantics it is often tedious and difficult.

Another line of research has focused on the development of automatic verification techniques for Solidity programs. For example, Mavridou et al. [31] provide an approach based on FSolidM [32], in which a Solidity smart contract is modeled as a state machine to support model checking of common security properties. In addition, Hajdu and Jovanovic [20] provide solc-verify, a modular verifier for Solidity smart contracts. Work in this area usually focuses on the automatic verification of different aspects of Solidity programs and can not be used to verify general *functional correctness*, which is the focus of our work.

Finally, some research has focused on the verification of functional correctness of Solidity programs. Early work in this area includes TinySol [7] and Featherweight Solidity [15], two calculi formalizing some of the core features of Solidity. More recently,

Ahrendt and Bubel described SolidiKeY [1], a formalization of a subset of Solidity in the KeY tool to verify data integrity for smart contracts. Similar to our work, research in this area can be used to verify the functional correctness of Solidity contracts. However, the above works differ from the work presented in this paper in two main aspects. First, the rules described above are provided in the form of axioms rather than being derived from a formal semantic, as is the case with our work. Second, the above works focus on a restricted subset of Solidity. For example, none of the works consider fixed-size integers, different types of stores with different semantics for assignments, or external vs. internal method calls, which are key features of Solidity addressed by our calculus.

8 Conclusion

In this paper, we presented a framework for the verification of Solidity smart contracts in Isabelle/HOL. To this end, we developed a calculus to reason about Solidity statements, formalized it in Isabelle, and mechanically verified its soundness. In addition, we developed a verification condition generator that automates the use of the calculus. To evaluate the approach, we used it to verify a basic token in Solidity, which showed that the calculus can significantly reduce the effort to verify Solidity smart contracts compared to a verification from its semantics.

While our calculus supports most of the important features of Solidity there are still some more advanced features of the language that are not yet supported. In particular, the calculus does not yet support inheritance, which seems to be an important feature for Solidity developers. Moreover, although our case study demonstrates the feasibility of our approach it is not clear how well it can be generalized to the verification of other contracts.

To address the above limitations, future work arises in two directions. First, future work should extend the calculus to support more advanced features of Solidity, such as inheritance. In addition, future work should also focus on conducting additional case studies in which the calculus is used for the verification of additional contracts.

Availability. Our formalisation and the evaluation results are available under BSD license (SPDX-License-Identifier: BSD-2-Clause) [29].

Acknowledgements. We would like to thank Achim Brucker for his support with Isabelle. Moreover, we would like to thank Wolfgang Ahrendt and Richard Bubel for inspiring discussions about the verification of Solidity contracts.

References

1. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 9–24. Springer International Publishing, Cham (2020)
2. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: A survey. Pervasive and Mobile Computing **67**, 101227 (2020). https://doi.org/10.1016/j.pmcj.2020.101227

3. Apt, K.R., de Boer, F., Olderog, E.R.: Verification of Sequential and Concurrent Programs. Springer Publishing Company, Incorporated, 3rd edn. (2009)
4. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on ethereum smart contracts sok. In: Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204. pp. 164–186. Springer-Verlag, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-662-54455-6_8
5. Azaria, A., Ekblaw, A., Vieira, T., Lippman, A.: Medrec: Using blockchain for medical data access and permission management. In: 2016 2nd International Conference on Open and Big Data (OBD). pp. 25–30 (2016). https://doi.org/10.1109/OBD.2016.11
6. Bahrynovska, T.: History of Ethereum Security Vulnerabilities, Hacks and Their Fixes. https://applicature.com/blog/blockchain-technology/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes, accessed: 2023-04-18
7. Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for Solidity contracts. In: Pérez-Solà, C., Navarro-Arribas, G., Biryukov, A., Garcia-Alfaro, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology. pp. 233–243. Springer (2019)
8. Batra, G., Olson, R., Pathak, S., Santhanam, N., Soundararajan, H.: Blockchain 2.0: What's in store for the two ends? https://www.mckinsey.com/industries/industrials-and-electronics/our-insights/blockchain-2-0-whats-in-store-for-the-two-ends-semiconductors-suppliers-and-industrials-consumers, accessed: 2023-04-18
9. Berghofer, S., Wenzel, M.: Inductive datatypes in hol — lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) TPHOLs. pp. 19–36. Springer (1999)
10. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Programming Languages and Analysis for Security. p. 91–96. PLAS, ACM (2016). https://doi.org/10.1145/2993600.2993611
11. Cassez, F., Fuller, J., Quiles, H.M.A.: Deductive verification of smart contracts with dafny. In: Groote, J.F., Huisman, M. (eds.) Formal Methods for Industrial Critical Systems. pp. 50–66. Springer International Publishing, Cham (2022)
12. Chavez-Dreyfuss, G.: Sweden tests blockchain technology for land registry. https://www.reuters.com/article/us-sweden-blockchain-idUSKCN0Z22KV, accessed: 2023-04-18
13. Clegg, P., Jevans, D.: Cryptocurrency crime and anti-money laundering report. Tech. rep., CipherTrace (2021)
14. Cock, D., Klein, G., Sewell, T.: Secure microkernels, state monads and scalable refinement. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics. pp. 167–182. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
15. Crafa, S., Di Pirro, M., Zucca, E.: Is Solidity solid enough? In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 138–153. Springer (2020)
16. Crosara, M., Centurino, G., Arceri, V.: Towards an Operational Semantics for Solidity. In: van Rooyen, J., Buro, S., Campion, M., Pasqua, M. (eds.) VALID. pp. 1–6. IARIA (Nov 2019)
17. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (aug 1975). https://doi.org/10.1145/360933.360975
18. Ethereum: Solidity. https://docs.soliditylang.org/, accessed: 2023-05-04
19. Gartner: Forecast blockchain business value, worldwide. https://www.gartner.com/en/documents/3627117 (2019), accessed: 2023-05-04

20. Hajdu, Á., Jovanovic, D.: solc-verify: A modular verifier for Solidity smart contracts. In: Chakraborty, S., Navas, J.A. (eds.) VSTTE. LNCS, vol. 12031, pp. 161–179. Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_11
21. Hajdu, Á., Jovanovic, D.: Smt-friendly formalization of the Solidity memory model. In: Müller, P. (ed.) ESOP. LNCS, vol. 12075, pp. 224–250. Springer (2020). https://doi.org/10.1007/978-3-030-44914-8_9
22. Jiao, J., Kan, S., Lin, S.W., Sanan, D., Liu, Y., Sun, J.: Semantic understanding of smart contracts: executable operational semantics of Solidity. In: SP. pp. 1695–1712. IEEE (2020)
23. Jiao, J., Lin, S.W., Sun, J.: A generalized formal semantic framework for smart contracts. In: Wehrheim, H., Cabot, J. (eds.) FASE. pp. 75–96. Springer (2020)
24. Kelly, J.: Banks adopting blockchain 'dramatically faster' than expected: IBM. https://www.reuters.com/article/us-tech-blockchain-ibm-idUSKCN11Y28D (2016), accessed: 2023-05-04
25. Llama, D.: Tvl breakdown by smart contract language. https://defillama.com/languages (2022)
26. Marmsoler, D., Brucker, A.D.: A Denotational Semantics Of Solidity In Isabelle/HOL. In: Software Engineering and Formal Methods: 19th International Conference, SEFM 2021, Virtual Event, December 6–10, 2021, Proceedings. pp. 403–422. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-92124-8_23
27. Marmsoler, D., Brucker, A.D.: Conformance Testing of Formal Semantics Using Grammar-Based Fuzzing. In: Kovács, L., Meinke, K. (eds.) Tests and Proofs. pp. 106–125. Springer International Publishing, Cham (2022)
28. Marmsoler, D., Brucker, A.D.: Isabelle/solidity: A deep embedding of solidity in isabelle/hol. Archive of Formal Proofs (July 2022), https://isa-afp.org/entries/Solidity.html, Formal proof development
29. Marmsoler, D., Thornton, B.: SSCalc - A Calculus for Solidity Smart Contracts (Apr 2023). https://doi.org/10.5281/zenodo.7846232
30. Matichuk, D., Wenzel, M., Murray, T.: An isabelle proof method language. In: Klein, G., Gamboa, R. (eds.) Interactive Theorem Proving. pp. 390–405. Springer International Publishing, Cham (2014)
31. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for Ethereum. In: FC (2019)
32. Mavridou, A., Laszka, A.: Tool demonstration: Fsolidm for designing secure Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) Principles of Security and Trust. pp. 270–277. Springer (2018)
33. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008)
34. News, B.: Hackers steal $600m in major cryptocurrency heist. https://www.securityweek.com/hackers-steal-over-600m-major-crypto-heist (2021), accessed: 2023-05-04
35. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic (2002)
36. Perez, D., Livshits, B.: Smart contract vulnerabilities: Vulnerable does not imply exploited. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 1325–1341. USENIX Association (Aug 2021)
37. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004), version 8.0
38. TNW: These are the top 10 programming languages in blockchain. https://thenextweb.com/news/javascript-programming-java-cryptocurrency (2019), accessed: 2023-05-04
39. Vogelsteller, F., Buterin, V.: "erc-20: Token standard", ethereum improvement proposals, no. 20. https://eips.ethereum.org/EIPS/eip-20 (11 2015)

40. Wadler, P.: Monads for functional programming. In: Broy, M. (ed.) Program Design Calculi. pp. 233–264. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
41. Yang, Z., Lei, H.: Lolisa: Formal syntax and semantics for a subset of the Solidity programming language in mathematical tool Coq. Mathematical Problems in Engineering **2020**, 6191537 (2020)
42. YCharts.com: Ethereum transactions per day. https://ycharts.com/indicators/ethereum_transactions_per_day (2022), accessed: 2023-05-04
43. Yurcan, B.: How blockchain fits into the future of digital identity (2016)
44. Zakrzewski, J.: Towards verification of Ethereum smart contracts. In: Piskac, R., Rümmer, P. (eds.) VSTTE. LNCS, vol. 11294, pp. 229–247. Springer (2018). https://doi.org/10.1007/978-3-030-03592-1_13