A Denotational Semantics of Solidity in Isabelle/HOL

Diego Marmsoler$^{1\,[0000-0003-2859-7673]}$ and Achim D. Brucker$^{1\,[0000-0002-6355-1200]}$

University of Exeter, Exeter, UK
{d.marmsoler, a.brucker}@exeter.ac.uk

Abstract. Smart contracts are programs, usually automating legal agreements such as financial transactions. Thus, bugs in smart contracts can lead to large financial losses. For example, an incorrectly initialized contract was the root cause of the Parity Wallet bug that made USD 280mil worth of Ether inaccessible. Ether is the cryptocurrency of the Ethereum blockchain that uses Solidity for expressing smart contracts. In this paper, we address this problem by presenting an executable denotational semantics for Solidity in the interactive theorem prover Isabelle/HOL. This formal semantics builds the foundation of an interactive program verification environment for Solidity programs and allows for inspecting Solidity programs by (symbolic) execution. We combine the latter with grammar-based fuzzing to ensure that our formal semantics complies to the Solidity implementation on the Ethereum Blockchain. Finally, we demonstrate the formal verification of Solidity programs by two examples: constant folding and memory optimization.

Keywords: Solidity · Denotational Semantics · Isabelle/HOL · Gas Optimization.

1 Introduction

An increasing number of businesses are adopting blockchain-based solutions. Notably, the market value of Bitcoin, most likely the first and most well-known blockchain-based cryptocurrency, passed USD 1 trillion in February 2021 [1]. While Bitcoin might be the most well-known application of a blockchain, it lacks features that applications outside of cryptocurrencies require and that make blockchain solutions attractive to businesses.

The Ethereum blockchain [40] is a feature-rich distributed computing platform that provides not only a cryptocurrency, called *Ether*: Ethereum also provides an immutable distributed data structure (the *blockchain*) on which distributed programs, called *smart contracts*, can be executed. Essentially, smart contracts are programs, usually automating legal agreements, e.g., financial transactions. To support such applications, Ethereum provides a dedicated account data structure on its blockchain that smart contracts can modify, i.e., transferring Ether between accounts. Thus, bugs in smart contracts can lead to large financial losses. For example, an incorrectly initialized contract was the root cause of the Parity Wallet bug that froze USD 280mil worth of Ether [32]. This risk of bugs being costly is already a big motivation for using formal verification techniques. The fact that smart contracts are deployed on the blockchain immutably, i.e., they cannot be updated or removed easily, makes it even more important to "get smart contracts right", before they are deployed on a blockchain for the very first time.

For implementing smart contracts, Ethereum provides *Solidity* [30], a Turing-complete, statically typed programming language that has been designed to look familiar to people knowing Java, C, or JavaScript. The following shows a simple (artificial) function of a smart contract in Solidity for withdrawing Ether:

```
function wd(uint256 n, address payable r) public returns(bool) {
    if (n < address(this).balance) {
      r.transfer(n);
      return true;
    }
    return false;
}
```

The type system provides, e.g., numerous integer types of different sizes (e.g., `uint256`) and the Solidity programs can make use of different types of stores for data (e.g., storage and memory). While Solidity is Turing-complete, the execution of Solidity programs is guaranteed to terminate. The reason for this is that executing Solidity operations costs *gas*, a tradeable commodity on the Ethereum blockchain. Gas does cost Ether and hence, programmers of smart contracts have an incentive to write highly optimized contracts whose execution consumes as little gas as possible. For example, the size of the integer types used can impact the amount of gas required for executing a contract. Similarly, different type of stores induce different gas costs. Thus, the authors of Solidity contracts try to optimize the costs of executing a contract. This desire for highly optimized contracts can conflict with the desire to write correct and secure contracts.

We address the problem of developing smart contracts in Solidity that are correct: we present an executable denotational semantics for Solidity in the interactive theorem prover Isabelle/HOL [28]. Our contributions are four-fold:

1. A formal semantics of (a subset of) Solidity as conservative embedding into Isabelle/HOL. We follow the LCF-approach [15] and do not use any axiomatic definitions and, hence, our semantics is consistent "by construction".
2. A grammar-based fuzzing framework that can automatically validate our formal semantics against the Ethereum blockchain. Thus, we can provide strong evidence that our formal semantics complies to the official implementation.
3. We use our formal semantics for building an integrated verification and symbolic execution environment for Solidity programs on top of Isabelle/HOL. For this, we developed domain-specific automated proof methods.
4. We showcase our verification approach by formally analyzing two optimization strategies from which we derive rules that can be used to optimize the gas consumption of Solidity programs while preserving their semantics.

Our approach combines an expressive logic, i.e., higher-order logic (HOL) within an interactive theorem prover with a testing framework allowing us to validate the formalization against the actual implementation. This combination enables us to quickly analyze the impact of changes to the semantics while ensuring formal consistency and compliance to the implementation. The ability to quickly assess changes in Solidity is important, as Solidity is a fast evolving language. The Solidity manual [30], e.g., states: "When deploying contracts, you should use the latest released version of Solidity. This is because breaking changes as well as new features and bug fixes are introduced regularly."

2 Semantics

In the following, we describe our denotational semantics for a subset of Solidity v0.5.16 [30][1]. The complete semantics is formalized in Isabelle/HOL [28]. The formalization consists of 1 500 lines of Isabelle code.

Our subset supports the following features of Solidity:

- *Fixed-size integer types* of various lengths and corresponding arithmetic with support for overflows.
- *Domain-specific primitives*, such as money transfer or balance queries.
- *Different types of stores*, such as storage, memory, and stack.
- *Complex data types*, such as hash-maps and arrays.
- *Assignments with different semantics*, depending on the location of the involved data types.
- An extendable *gas model*.

Our formalization is based on higher-order logic using inductive datatypes [7]. To this end, we use **bold** font for types and *italics* for type constructors.

2.1 Value types

Solidity supports four different basic data types, called *value types*:

$$\mathbf{Types} \quad ::= \quad \mathit{TBool} \mid \mathit{TAddr} \mid \mathit{TSInt}\ \mathbf{Nat} \mid \mathit{TUInt}\ \mathbf{Nat}$$

TBool denotes boolean values and *TAddr* denotes addresses. Solidity also supports signed and unsigned integers from 8 to 256 bits in steps of 8. Thus, *TSInt b* and *TUInt b* denote signed and unsigned integers of 2^b bit size.

In Solidity, raw data is encoded in hexadecimal format, however, to simplify the computation of locations for reference types (as discussed in more detail in Sect. 2.2), we use strings to model raw data in our model. Thus, type **Valuetype** is actually just a synonym for type string and it is used to represent the data of value types in the store. In addition, we shall write $\lfloor v \rfloor$ and $\lceil v \rceil$ to convert the value v of a basic data type to and from a string representation, respectively.

Converting an integer to a corresponding bit representation can result in an overflow which needs to be considered. Thus, we define two functions *createSInt*

[1] This is the currently supported default version of the Truffle test framework.

and *createUInt* to convert an arbitrary number to a corresponding signed or unsigned integer representation of a certain size:

$$createSInt\colon \mathbf{Nat} \times \mathbf{Int} \to \mathbf{Valuetype}$$

$$createSInt(b, v) = \begin{cases} \left\lfloor \left((v + 2^{b-1}) \bmod 2^b\right) - 2^{b-1} \right\rfloor & \textit{if } v \geq 0 \\ \left\lfloor 2^{b-1} - \left((2^{b-1} - v - 1) \bmod 2^b\right) - 1 \right\rfloor & \textit{if } v < 0 \end{cases}$$

where x mod y denotes the non-negative remainder when dividing x by y. The definition of *createUInt* is similar.

Essentially, the functions can be used to create a representation of a given number which fits into a certain bit size. For example, $createSInt(8, 200) =$ "-56" whereas $createUInt(8, 200) =$ "200".

We can then define functions to lift basic arithmetic and boolean operations to corresponding operations over signed and unsigned integers of various sizes. The operation *add*, for example, can be defined by the following equations using usual pattern-matching notation:

$$add\colon \mathbf{Types} \times \mathbf{Types} \times \mathbf{Valuetype} \times \mathbf{Valuetype} \to (\mathbf{Valuetype} \times \mathbf{Types})_\perp$$

$$add\,(TUInt(b_l), TUInt(b_r), v_l, v_r) = createU\,(max(b_l, b_r), \lceil v_l \rceil + \lceil v_r \rceil)$$

$$add\,(TSInt(b_l), TSInt(b_r), v_l, v_r) = createS\,(max(b_l, b_r), \lceil v_l \rceil + \lceil v_r \rceil)$$

$$add\,(TUInt(b_l), TSInt(b_r), v_l, v_r) = \begin{cases} createS\,(b_r, \lceil v_l \rceil + \lceil v_r \rceil) & \text{if } b_l < b_r \\ \perp & \text{if } b_l \geq b_r \end{cases}$$

$$add\,(TSInt(b_l), TUInt(b_r), v_l, v_r) = \begin{cases} createS\,(b_l, \lceil v_l \rceil + \lceil v_r \rceil) & \text{if } b_r < b_l \\ \perp & \text{if } b_r \geq b_l \end{cases}$$

where $createU(b, v) = (createUInt(b, v), TUInt(b))$, and
$$createS(b, v) = (createSInt(b, v), TSInt(b))\,.$$

According to the current specification of Solidity, adding two integers of the same type is always possible but results in a new integer of the size of the larger one. Adding integers of different type is only possible if the size of the signed integer is strictly greater than the one of the unsigned one, in which case the result is always a signed integer with the size of the signed one. Moreover, the result of adding two numbers might not fit into the corresponding result type in which case an overflow occurs.

Consider, e.g., the following two additions of an unsigned with a signed integer:

$$add(TUInt(8), TSInt(16), \text{"200"}, \text{"32600"}) = (\text{"-32736"}, TSInt(16))$$

$$add(TUInt(16), TSInt(16), \text{"100"}, \text{"32700"}) = \perp$$

In the first case, $32600 + 200$ does not fit into the resulting 16-bit signed integer (which can only store numbers up to 32767) and thus we get an overflow. In the second case, we try to add two incompatible types which results in an error. Similar definitions can be provided for the remaining arithmetic and logical operators.

2.2 Stores and Reference Types

In Solidity, storage cells are addressed by hexadecimal numbers. Again, however, we use strings to model them to simplify computation of locations for reference types. Thus, type **Loc** denotes the type of strings and is used to represent storage locations. We can then model a general store for values of type v as a parametric data type:

$$\textbf{Store}\ v \quad ::= \quad (\textbf{Loc} \rightarrow v) \times \textbf{Nat}$$

It consists of a mapping to assign values to locations and in addition it holds a pointer to the next free location. We can then define function $access(l, s)$ to access the value at location l in store s and function $updateStore(l, v, s)$ to store value v at location l of store s. The definition of these functions is standard and thus not discussed further. However, the way Solidity computes storage locations for reference types is a bit special and thus worth a closer look. To this end, assume that a storage cell loc contains a reference type, such as a mapping. Then, the storage cell which contains the value of the entry for key k is computed by $keccak256(\text{“}k\text{”} + loc)$, where $keccak256$ denotes the Keccak hash function [8] and + denotes string concatenation.

The main objective of this approach is to obtain a unique storage cell for every element. The purpose of using the hash value is to deal with a limited amount of storage cells which are available in practice. In theory, collisions are possible when using a hash function, however, in practice, such collisions are very unlikely to happen and thus they may be neglected. Thus, in our model, the location of the storage cell which holds the value of an element ix of a reference type which is stored at location loc is obtained by concatenating ix with loc separated by a dot:

$$h(loc,\ ix) = ix + \text{“.”} + loc$$

Types of Storage. Solidity has three different stores: stack, memory, and storage. The *stack* stores the values for variables which can either be concrete values (for value type variables) or pointers to either memory or storage (for reference type variables). Thus, a stack can be modelled as a store which can keep three different types of values:

$$\begin{aligned} \textbf{Stackvalue} &::= \mathit{Value}\ \textbf{Valuetype} \mid \mathit{Memptr}\ \textbf{Loc} \mid \mathit{Stoptr}\ \textbf{Loc} \\ \textbf{Stack} &::= \textbf{Store Stackvalue} \end{aligned}$$

Solidity supports two additional stores *memory* and *storage* for storing the value of *reference types*. While memory supports only arrays, storage also supports mappings:

$$\begin{aligned} \textbf{MTypes} &::= \mathit{MTValue}\ \textbf{Types} \mid \mathit{MTArray}\ \textbf{Nat}\ \textbf{MTypes} \\ \textbf{STypes} &::= \mathit{STValue}\ \textbf{Types} \mid \mathit{STArray}\ \textbf{Nat}\ \textbf{STypes} \mid \mathit{STMap}\ \textbf{Types}\ \textbf{STypes} \end{aligned}$$

The internal organization of the two stores differs fundamentally: While memory uses pointer structures to organize the values of reference types, storage values are accessed directly by computing the corresponding location. Thus we model memory as a store which can keep two different types of values:

$$\begin{aligned}\textbf{Memoryvalue} &::= \textit{Value}\ \textbf{Valuetype} \mid \textit{Pointer}\ \textbf{Loc}\\ \textbf{Memory} &::= \textbf{Store Memoryvalue}\end{aligned}$$

Storage, on the other hand is modeled as a simple store of value types:

$$\textbf{Storage} ::= \textbf{Store Valuetype}$$

Storage access is non-strict, which means that access to an undefined storage cell returns a default value. To this end, we first define a function $\mathit{ival}\colon \textbf{Types} \to \textbf{Valuetype}$ which returns a default value for each value type. Now, we can define a corresponding access function for storage:

$$\mathit{accessStorage}\colon \textbf{Types} \times \textbf{Loc} \times \textbf{Storage} \to \textbf{Valuetype}$$

$$\mathit{accessStorage}(t, \mathit{loc}, \mathit{sto}) = \begin{cases} v, & \text{if } v \neq \bot \\ \mathit{ival}(t), & \text{if } v = \bot \end{cases} \quad \text{where } v = \mathit{access}(\mathit{loc}, \mathit{sto})$$

Copying of Reference Types. Often, we need to copy values from one type of store to another, i.e., we need different types of copy functions. To specify them, we use a higher-order function

$$\mathit{iter} : (\textbf{Int} \to a \to a) \to a \to \textbf{Int} \to a$$

such that $\mathit{iter}(f, x, v)$ executes function f on value v and the passes the outcome on to another execution of f until f was executed x times.

In the following we use *iter* to define the function to copy from storage to memory:

$$\mathit{cp}^s_m\colon \textbf{Loc} \times \textbf{Loc} \times \textbf{Int} \times \textbf{STypes} \times \textbf{Storage} \times \textbf{Memory} \to \textbf{Memory}_\bot$$

$$\mathit{cp}^s_m(l_s, l_m, i, t, s, m) = \mathit{iter}(\lambda i', m'.\ \mathit{cprec}^s_m(h(l_s, \lfloor i' \rfloor), h(l_m, \lfloor i' \rfloor), t, s, m'), m, i)$$

$$\begin{aligned}\text{where}\quad & \mathit{cprec}^s_m(l_s, l_m, \mathit{STArray}(i, t), s, m) = \\ & \quad \mathit{iter}\ (\lambda i', m'.\ \mathit{cprec}^s_m(h(l_s, \lfloor i' \rfloor), h(l_m, \lfloor i' \rfloor), t, s, m'), m'', i) \\ & \quad \text{where } m'' = \mathit{updateStore}(l_m, \mathit{Pointer}(l_m), m) && (1)\\ & \mathit{cprec}^s_m(l_s, l_m, \mathit{STValue}(t), s, m) = \mathit{updateStore}(l_m, \mathit{Value}(v), m) \\ & \quad \text{where } v = \mathit{accessStorage}(t, l_s, s) && (2)\\ & \mathit{cprec}^s_m(l_s, l_m, \mathit{STMap}(t, t'), s, m) = \bot && (3)\end{aligned}$$

In Solidity, value types are just copied between stores which is reflected by Eq. (2). For reference types, however, the situation is different. Mappings can only be kept in storage and not in memory which is why a mapping is never copied from storage to memory, and we just return $\bot$ for this case (Eq. (3)). Arrays, on the other hand, can be kept in both: storage and memory. As mentioned above, however, the way of storing them differs depending on the type of store: in storage, we just calculate the location of the elements of an array whereas in memory arrays are stored using a pointer structure. Thus, when copying arrays from storage to memory we need to create the corresponding pointer structure as shown by Eq. 1.

Our model contains similar functions to copy from memory to storage or storage to storage. Copying from memory to memory is not required since memory operations do not copy the data structure but rather just the pointer as discussed in more detail in Sect. 2.4). It also contains similar functions to copy from memory to storage or storage to storage. Copying from memory to memory is not required since memory operations do not copy the data structure but rather just the pointer (see Sect. 2.4).

State. Accounts are associated with an address in hexadecimal format. We model **Address** as strings and accounts as mappings from addresses to their balance:

$$\mathbf{Accounts} ::= \mathbf{Address} \rightarrow \mathbf{Valuetype}$$

A state of a Solidity program consists of the balances of the accounts as well as the state of the different stores:

$$\mathbf{State} ::= \mathbf{Accounts} \times \mathbf{Stack} \times \mathbf{Memory} \times \mathbf{Storage}$$

In the following we shall use $\mathit{sck}(s)$, $\mathit{mem}(s)$, $\mathit{sto}(s)$, $\mathit{acc}(s)$ to access the stack, memory, storage, and account of a state s. Moreover, we use $\mathit{upSck}(k, s)$, $\mathit{upMem}(m, s)$, $\mathit{upSto}(t, s)$, and $\mathit{upAcc}(a, s)$ to change stack, memory, storage, or account, of a state s to k, m, t, or a, respectively.

2.3 Expressions

Our subset of Solidity supports basic arithmetic and boolean expressions over signed and unsigned integers of various bit sizes:

$$\begin{array}{lll}
\mathbf{B} & ::= & 8 \mid 16 \mid \ldots \mid 256 \\
\mathbf{L} & ::= & \mathit{Id}\ \mathbf{S} \mid \mathit{Ref}\ \mathbf{S}\ [\mathbf{E}] \\
\mathbf{E} & ::= & \mathit{Address}\ \mathbf{S} \mid \mathit{Balance}\ \mathbf{S} \mid L\ \mathbf{L} \mid \mathit{SInt}\ \mathbf{B}\ \mathbf{Int} \mid \mathit{UInt}\ \mathbf{B}\ \mathbf{Int} \mid \mathit{True} \mid \mathit{False} \\
& & \mid \mathbf{E} == \mathbf{E} \mid \mathbf{E} + \mathbf{E} \mid \mathbf{E} - \mathbf{E} \mid \mathbf{E} < \mathbf{E} \mid \neg\mathbf{E} \mid \mathbf{E} \wedge \mathbf{E} \mid \mathbf{E} \vee \mathbf{E}
\end{array}$$

where **S** denotes the type of strings, **Int** the type of integer symbols, and $[a]$ a list of elements of type a.

Environment. Expressions are always interpreted w.r.t. an environment which assigns types and values to variables. To this end, we introduce a new type **Identifier** (a synonym of type string) for variable names. Variables in Solidity can either be storage references or stack references which can again be pointers to either storage or memory. In addition, the environment also contains the address of the currently executing contract:

$$\begin{aligned}\textbf{Type} &::= \textit{Value}\ \textbf{Types} \mid \textit{Memory}\ \textbf{MTypes} \mid \textit{Storage}\ \textbf{STypes}\\ \textbf{Denvalue} &::= \textit{Stackloc}\ \textbf{Loc} \mid \textit{Storeloc}\ \textbf{Loc}\\ \textbf{Environment} &::= \textbf{Address} \times (\textbf{Identifier} \to \textbf{Type} \times \textbf{Denvalue})\end{aligned}$$

Lookup functions. To access the value of a reference type we define a function which looks up the corresponding value in memory:

$$\mathcal{M}\colon [\mathbf{E}] \to \mathbf{MTypes} \to \mathbf{Loc} \to \mathbf{Environment} \to \mathbf{State} \to \mathbf{Loc} \times \mathbf{MTypes}_{\bot}$$

$$\mathcal{M}[\![x]\!]t\ l\ e\ s\ = \begin{cases}(h(l,v),t') & \text{if } \textit{lookup}(x,t,e,s,t',v)\\ \bot & \text{otherwise}\end{cases}$$

$$\mathcal{M}[\![x\#xs]\!]t\ l\ e\ s\ = \begin{cases}\mathcal{M}[\![xs]\!]t'\ l'\ e\ s & \text{if } \textit{lookup}(x,t,e,s,t',v)\\ & \wedge\ \textit{access}(h(l,v),\textit{mem}(s)) = \textit{Pointer}(l')\\ \bot & \text{otherwise}\end{cases}$$

$$\begin{aligned}\text{where } \textit{lookup}(x,t,e,s,t',v) \iff\ & \exists lg, t''\colon t = \textit{MTArray}(lg,t')\\ & \wedge\ \mathcal{E}[\![x]\!]e\ s = (\textit{Value}(v), \textit{Value}(t''))\\ & \wedge\ \textit{less}(t'', \textit{TUInt}(256), v, \lfloor lg \rfloor) = (\text{“}\textit{True}\text{”}, \textit{TBool})\end{aligned}$$

Since memory uses pointer structures, we need to access the memory in every iteration to look up the next location.

Let us assume that $t = \textit{MTArray}(5, \textit{MTArray}(6, \textit{MTValue}(\textit{TBool})))$, and the memory of state s is $[\text{“3.2”} \mapsto \textit{Pointer}(\text{“5”}), \text{“4.5”} \mapsto \textit{Value}(\text{“}\textit{True}\text{”})]$. Then,

$$\mathcal{M}[\![[\textit{UInt}(8,3)]]\!]t\ \text{“2”}\ e\ s = (\text{“3.2”}, \textit{MTArray}(6, \textit{MTValue}(\textit{TBool}))) \quad (4)$$

$$\mathcal{M}[\![[\textit{UInt}(8,3), \textit{SInt}(8,4)]]\!]t\ \text{“2”}\ e\ s = (\text{“4.5”}, \textit{MTValue}(\textit{TBool})) \quad (5)$$

$$\mathcal{M}[\![[\textit{UInt}(8,5)]]\!]t\ \text{“2”}\ e\ s = \bot \quad (6)$$

$$\mathcal{M}[\![[\textit{UInt}(8,2)]]\!]t\ \text{“2”}\ e\ s = \bot \quad (7)$$

A similar function to $\mathcal{M}$ is defined to look up storage values with two notable differences:

- Since storage does not support pointer structures, we do not access the store while iterating through the list of selectors. Thus, the function always returns a storage location as long as we access indices within the range of the array (Eq. (7), for example would return a valid storage location).
- Since storage also supports mappings, the function can be used to look up also the value for mapping variables.

Using these functions we can then define two additional functions to look up the value or location of a variable:

$$\begin{aligned}&\mathcal{R}\colon \mathbf{L} \to \mathbf{Environment} \to \mathbf{State} \to \mathbf{Stackvalue} \times \mathbf{Type}_{\bot}\\ &\mathcal{L}\colon \mathbf{L} \to \mathbf{Environment} \to \mathbf{State} \to \mathbf{LType} \times \mathbf{Type}_{\bot}\\ &\text{with } \mathbf{LType}\ ::=\ \textit{Stackloc}\ \mathbf{Loc} \mid \textit{Memloc}\ \mathbf{Loc} \mid \textit{Storeloc}\ \mathbf{Loc}\end{aligned}$$

The definition of these functions is straightforward using the lookup functions discussed before and not discussed further here.

Semantics of Expressions. Finally we can define the semantic function for expressions.

$$\mathcal{E}\colon \mathbf{E} \to \mathbf{Environment} \to \mathbf{State} \to \mathbf{Stackvalue} \times \mathbf{Type}_{\perp}$$

The definition of the function mainly follows traditional denotational semantics definitions [34,35] with the exception that we use the operators introduced in Sect. 2.1 to manipulate integers:

$$\mathcal{E}[\![\mathit{SInt}(b,n)]\!]e\ s = (\mathit{Value}(\mathit{createSInt}(b,n)),\ \mathit{Value}(\mathit{TSInt}(b)))$$

$$\mathcal{E}[\![x_1 + x_2]\!]e\ s = \begin{cases} (\mathit{Value}(v),\ \mathit{Value}(t)) & \text{if } \mathcal{E}[\![x_1]\!]e\ s = (\mathit{Value}(v_1),\ \mathit{Value}(t_1)) \\ & \wedge\ \mathcal{E}[\![x_2]\!]e\ s = (\mathit{Value}(v_2),\ \mathit{Value}(t_2)) \\ & \wedge\ \mathit{add}(t_1, t_2, v_1, v_2) = (v, t) \\ \perp & \text{otherwise} \end{cases}$$

2.4 Statements

So far, our subset of Solidity supports variable declarations with optional initialisation and basic programming language statements:

$$\begin{aligned} \mathbf{D}\ &::= \mathbf{S} \times \mathbf{Type} \times \mathbf{E}_{\perp} \\ \mathbf{C}\ &::= \mathit{Skip} \mid \mathbf{L} = \mathbf{E} \mid \mathbf{C}\ ;\ \mathbf{C} \mid \mathit{Ite}\ \mathbf{E}\ \mathbf{C}\ \mathbf{C} \mid \mathit{While}\ \mathbf{E}\ \mathbf{C} \mid \mathit{Transfer}\ \mathbf{S}\ \mathbf{E} \mid \\ &\quad\ \mathit{Block}\ \mathbf{D}\ \mathbf{C} \end{aligned}$$

We can then define a semantic function for statements:

$$\mathcal{C}\colon\ \mathbf{C} \to \mathbf{Environment} \to \mathbf{State} \to \mathbf{Nat} \to (\mathbf{State} \times \mathbf{Nat})_{\perp}$$

The definition of it is mostly standard denotational semantics with some exceptions discussed in the following.

Gas. One interesting aspect of Solidity is that execution of statements is subject to fees, i.e., the execution consumes gas: if all gas is consumed, the execution terminates with an exception. Consequently, Solidity programs always terminate. The actual gas fees are computed on the level of the Ethereum byte code [39] and, moreover, are frequently updated. Thus, our Solidity formalization does not provide a built-in gas model trying to faithfully represent the actual gas model on the level of Ethereum bytecode: we only assume the existence of a generic cost function $\mathit{costs}\colon \mathbf{C} \times \mathbf{Environment} \times \mathbf{State} \to \mathbf{N}$ which provides the gas costs for executing a given statement. A separate gas function for expressions can be defined and used with the cost function for statements. Moreover, in our subset of Solidity, the while statement is the only program statement that does not terminate in all states. Therefore, we require:

$$0 < \mathit{costs}(\mathit{While}(ex, s), e, s') \tag{8}$$

This requirement is not a limitation, as the actual costs for any execution of a while loop will be positive [39, Appendix G]. While our cost model can, in principle, be used for proving upper or lower bounds for the gas consumption of a given contract, the usefulness of such a statement depends on how faithful the user-provided cost functions model the actual costs which may also depend on compiler optimizations.

We can now verify a general statement about the semantics, namely that it always terminates. Note that we model error states (e.g., failing transfers) using an explicit error type. This is a standard construction to model partial functions in HOL, which requires that all functions are total from a "logical perspective."

Theorem 1. *$\mathcal{C}[\![c]\!]\ e\ s\ g$ is always defined.*

Proof. The proof is a simple inductive argument over c using Eq. (8). □

Indeed, Isabelle automatically proves it for us and provides us with corresponding proof methods to support reasoning over $\mathcal{C}$.

Semantics of Assignments. Another particularity of Solidity is that the semantics of an assignment depends on the type of store to which the involved variables refer. Let us consider, for example, the case in which the right-hand side of an assignment evaluates to a value stored in memory:

$$\mathcal{C}[\![v{=}x]\!]e\ s\ g$$
$$=\begin{cases}(1a) \quad \mathit{if}\ \ ex(g,x,e,s,p,i,t) \wedge \mathcal{L}[\![v]\!]e\ s{=}(\mathit{Stackloc}(l),\mathit{Memory}(t'))\\ (2a) \quad \mathit{if}\ \ ex(g,x,e,s,p,i,t) \wedge \mathcal{L}[\![v]\!]e\ s{=}(\mathit{Stackloc}(l),\mathit{Storage}(t'))\\ \ \wedge\ \mathit{access}(l,\mathit{sck}(s)){=}\mathit{Stoptr}(p') \wedge \mathit{cp}_s^m(p,p',i,t,\mathit{mem}(s),\mathit{sto}(s)){=}o\\ (3a) \quad \mathit{if}\ \ ex(g,x,e,s,p,i,t) \wedge \mathcal{L}[\![v]\!]e\ s{=}(\mathit{Storeloc}(l),t')\\ \ \wedge\ \mathit{cp}_s^m(p,l,i,t,\mathit{mem}(s),\mathit{sto}(s)){=}o\\ (4a) \quad \mathit{if}\ \ ex(g,x,e,s,p,i,t) \wedge \mathcal{L}[\![v]\!]e\ s{=}(\mathit{Memloc}(l),t')\\ \qquad\quad \ldots\end{cases}$$

$$\text{where } ex(g,x,e,s,p,i,t) \Longleftrightarrow \mathit{costs}(v{=}x,e,s){<}g$$
$$\wedge\ \mathcal{E}[\![x]\!]e\ s{=}(\mathit{Memptr}(p),\mathit{Memory}(\mathit{MTArray}(i,t)))$$
$$(1a){=}(\mathit{upSck}(\mathit{updateStore}(l,\mathit{Memptr}(p),\mathit{sck}(s)),s),\mathit{costs}(v{=}x,e,s))$$
$$(2a,3a){=}(\mathit{upSto}(o,s),\mathit{costs}(v{=}x,e,s))$$
$$(4a){=}(\mathit{upMem}(\mathit{updateStore}(l,\mathit{Pointer}(p),\mathit{mem}(s)),s),\mathit{costs}(v{=}x,e,s))$$

In this case, the semantics of the assignment changes, depending on the $\mathcal{L}$-value of the left-hand side: If it is a pointer to memory (cases (1) and (4)), we just assign the pointer but if it is a reference to storage (cases (2) and (3)), we copy the whole structure to memory using the copy functions discussed in Sect. 2.2.

Transferring Money. Another aspect which sets Solidity apart from traditional programming languages is its support for features to transfer funds from one account to another. To this end, every contract is associated with an account and Solidity supports a command which can be used to transfer funds from it to another account:

$$\mathcal{C}[\![\mathit{Transfer}\ a\ x]\!]e\ s\ g = \begin{cases} (1b) & \mathit{if}\ \ \mathit{costs}(\mathit{Transfer}(a,x),e,s) < g \\ & \wedge\ \mathcal{E}[\![x]\!]e\ s = (\mathit{Value}(v),\ \mathit{Value}(t)) \\ & \wedge\ \mathit{transfer}(\mathit{address}(e),a,t,v,\mathit{acc}(s)) = \mathit{ac} \\ \ldots \end{cases}$$

where $(1b) = (\mathit{upAcc}(\mathit{ac},s), \mathit{costs}(\mathit{Transfer}(a,x),e,s))$

$$\mathit{transfer}(s,d,t,v,\mathit{ac}) = \begin{cases} \mathit{addB}(d,t,v,\mathit{ac}') & \text{if } \mathit{subB}(s,t,v,\mathit{ac}) = \mathit{ac}' \\ \perp & \text{otherwise} \end{cases}$$

Here, $\mathit{address}(e)$ denotes the address of the contract's account, and $\mathit{addB}(a,t,v,\mathit{ac})$ and $\mathit{subB}(a,t,v,\mathit{ac})$ are functions to increase and decrease the balance of an address a of accounts ac by a certain amount v. Note that both functions use the corresponding add and sub functions for signed and unsigned integers discussed in Sect. 2.1. Moreover, subB may also fail if an account has not enough funds in which case it evaluates to $\perp$.

3 Compliance to the Official Solidity Implementation

For ensuring that our formal semantics is a faithful representation of the official Solidity implementation, we provide a test framework that supports comparing the result of evaluating a Solidity program in our formal semantics to its execution on the Ethereum blockchain.

We use Isabelle's code generator to automatically generate a Solidity evaluator from our formal semantics. In our current implementation, we use Haskell as target platform for the code generator. Moreover, we need to provide a concrete cost function for computing the gas consumption (recall Sect. 2.4). In Isabelle, we can achieve this by instantiating a so-called locale [5] with a trivial implementation satisfying Eq. (8).

We then generate Solidity programs using a grammar-based fuzzer and compare the results of executing those programs on both the reference implementation of Solidity and our evaluator. The test framework is fully automated.Fig. 1 shows the main steps of our test framework that we discuss in the following in more detail.

- **Generate Random Solidity Code.** The test framework generates a random Solidity program from a given grammar, using the grammar-based fuzzer Grammarinator [21]. To avoid the generation of programs which do not compile, the grammar needs to be strict to only accept programs which are type-correct. The grammar is given in the format used by ANTLR4 [31].

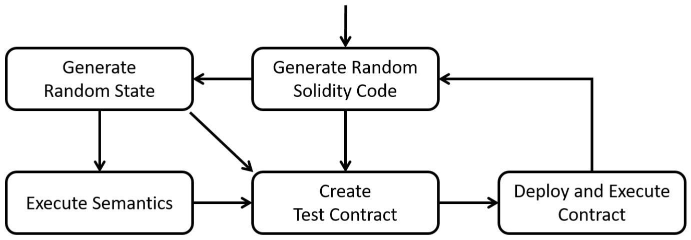

Fig. 1. Fuzzy testing Solidity smart contracts.

- **Generate Random State.** For each generated Solidity program, our testing framework generates a set of random input states. To this end, the script analyses the generated program and extracts the variables which occur in it. Based on the type of the variable, the script then generates random values for each variable.
- **Execute Semantics.** Before we can compute the output state with our semantics, we first need to transform the generated Solidity program to the abstract syntax which is accepted by the semantics. Finally, the abstract syntax of the program and the generated input state can be passed to the executable semantics, i.e., the evaluator automatically generated by Isabelle, to compute a corresponding output state.
- **Create Test Contract.** The generated Solidity program, together with the generated input state and the computed output state, is used to create a test contract for the Truffle testing framework [11]. Listing 1.1 shows parts of a generated contract, consisting of a single function which contains the generated Solidity program. The extracted storage variables are declared as contract variables whereas the extracted memory/stack variables are declared locally. Then, the variables are initialized according to the generated input state whereas the computed output state is used to create corresponding assertions for the Truffle framework.
- **Deploy and Execute Contracts.** Finally, the script deploys the test contract to a local instance of the Ganache blockchain [10] and executes the test using Truffle [11]. It then parses the output of the test, reports in in a log file and starts a new iteration.

3.1 Results

To test our semantics, we run the framework for several days which resulted in more than 10 000 successful tests. To cross-validate the effectiveness of the testing framework we also collected coverage information for the semantics using the Hpc tool [14]. The results are summarized in Fig. 2: Out of 123 definitions, 121 were executed during the tests. In addition, 186 alternatives (out of 524) and 1 592 expressions (out of 2 394) were executed. Hpc also generates detailed coverage reports for every module. When inspecting these reports it turns out that the low number of covered alternatives is mainly because of missing executions of error

```
contract TestContract0 {
    uint8 v_u8_s8;
    mapping(uint16 => uint8) v_m_u16_u8_9;          } Extracted
    bool[1][2] a_b_12_s5;                           } storage variables
    ...
    function test() public {
        uint104 v_u104_m2;                          } Extracted
        uint104[1][1] memory a_u104_11_m2;          } memory/stack variables
        ...
        v_u104_m2=14622709355569675963178665339646;   } Generated
        v_m_u16_u8_9[59381]=79;                        } input state
        ...
        int8 counter1=int8(0);
        while((v_m_u224_s240_1[uint224(444)]==
            (v_u216_s1-v_u104_m2)) && counter1<int8(10)){
            0xf7218C33533a3F22e3296F8b1DC0074B399355Eb        } Generated
                    .transfer(v_m_u16_u8_9[uint16(0)]);       } program
            counter1=counter1+int8(1);
        }
        ...
        Assert.equal(v_m_u16_u8_9[59381]==79, true);
        Assert.equal(a_u104_11_m2[0][0]==
            8130097819054169632795960896007, true);
        Assert.equal(                                    } Computed
            0xf7218C33533a3F22e3296F8b1DC0074B399355Eb   } result state
            .balance==100000000000000000000, true);
        ...
    }
}
```

Listing 1.1. Example test contract generated by our testing framework.

cases (e.g. ill-typed programs). This is because the test framework only generates well-formed Solidity programs and thus the error cases are not executed.

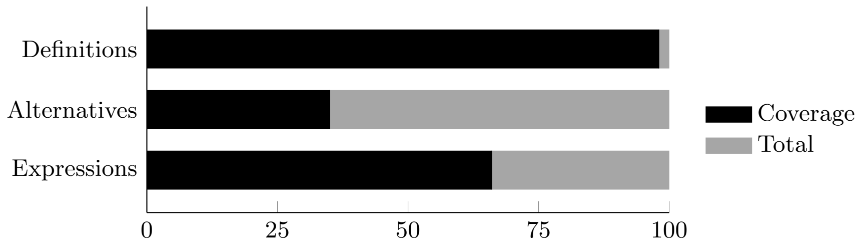

Fig. 2. Overall test coverage of semantics.

4 Verified Constant Folding

Constant folding is a common type of program optimization technique in which constant sub-expressions are replaced by their value. For example, the expression $\mathit{SInt}(16, 250) + \mathit{UInt}(8, 500)$ can be replaced with the expression $\mathit{SInt}(16, 494)$ in every program without affecting its outcome.

When it comes to smart contracts, constant folding is a good candidate for gas optimization. For example, according to the Remix IDE [29], computing the original expression costs 20 gas whereas computing the optimized version costs only 8 gas which leads to a saving of 12 gas just for this simple expression.

We can define a function for constant folding of Solidity expressions as follows:

$$\mathit{update}\colon \mathbf{E}{\to}\mathbf{E}$$

$$\mathit{update}(\mathit{SInt}(b,v))=\begin{cases}\mathit{SInt}\big(b,\big((v{+}2^{b-1}) \bmod 2^{b}\big){-}2^{b-1}\big) & \mathit{if}\ v{\geq}0\\ \mathit{SInt}\big(b,2^{b-1}{-}\big((2^{b-1}{-}v{-}1) \bmod 2^{b}\big){-}1\big) & \mathit{if}\ v{<}0\end{cases}$$

$$\mathit{update}(x_1{+}x_2)=\begin{cases}(1c) & \mathit{if}\ \exists b_1,v_1,b_2,v_2.\ \mathit{sint}(x_1,b_1,v_1,x_2,b_2,v_2)\ \wedge\ v_1{+}v_2{\geq}0\\ (2c) & \mathit{if}\ \exists b_1,v_1,b_2,v_2.\ \mathit{sint}(x_1,b_1,v_1,x_2,b_2,v_2)\ \wedge\ v_1{+}v_2{<}0\\ (3c) & \mathit{if}\ \exists b_2{<}b_1,v_1,v_2.\ \mathit{uint}(x_1,b_1,v_1,x_2,b_2,v_2)\ \wedge\ v_1{+}v_2{\geq}0\\ (4c) & \mathit{if}\ \exists b_2{<}b_1,v_1,v_2.\ \mathit{uint}(x_1,b_1,v_1,x_2,b_2,v_2)\ \wedge\ v_1{+}v_2{<}0\\ \ldots\end{cases}$$

with

$$\mathit{sint}(x_1,b_1,v_1,x_2,b_2,v_2){\iff}\mathit{update}(x_1){=}\mathit{SInt}(b_1,v_1)\ \wedge\ \mathit{update}(x_2){=}\mathit{SInt}(b_2,v_2)$$

$$\mathit{uint}(x_1,b_1,v_1,x_2,b_2,v_2){\iff}\mathit{update}(x_1){=}\mathit{SInt}(b_1,v_1)\ \wedge\ \mathit{update}(x_2){=}\mathit{UInt}(b_2,v_2)$$

$$(1c){=}\mathit{SInt}\Big(\mathit{max}(b_1,b_2),\Big((2^{\mathit{max}(b_1,b_2)-1}{+}v) \bmod 2^{\mathit{max}(b_1,b_2)}\Big){-}2^{\mathit{max}(b_1,b_2)-1}\Big)$$

$$(2c)$$

$$={\mathit{SInt}}\Big(\mathit{max}(b_1,b_2),2^{\mathit{max}(b_1,b_2)-1}{-}\Big((2^{\mathit{max}(b_1,b_2)-1}{-}v{-}1) \bmod 2^{\mathit{max}(b_1,b_2)}\Big){-}1\Big)$$

$$(3c){=}\mathit{SInt}\big(b_1,\big((v{+}2^{b_1-1}) \bmod 2^{b_1}\big){-}2^{b_1-1}\big)$$

$$(4c){=}\mathit{SInt}\big(b_1,2^{b_1-1}{-}\big((2^{b_1-1}{-}v{-}1) \bmod 2^{b_1}\big){-}1\big)$$

where for every case $(1c){-}(4c)$, variables b_1,v_1,b_2,v_2 denote the *unique* elements satisfying the condition required for this case and $v{=}v_1{+}v_2$. The cases for unsigned integers and the remaining arithmetic and boolean expressions are similar.

The function *update* can be applied to a Solidity program to replace constant expressions with their corresponding value reducing the gas cost of executing the program. For example, *update* applied to the expression $\mathit{SInt}(16, 250) + \mathit{UInt}(8, 500)$ returns the expression $\mathit{SInt}(16, 494)$.

Having a formal semantics of Solidity expressions in Isabelle allows us to mechanically verify the correctness of our *update* function, i.e., we proved in Isabelle/Isar [38] that it does not modify the semantics of an expression:

Theorem 2. $\mathcal{E}[\![x]\!]e\ s = \mathcal{E}[\![\mathit{update}(x)]\!]e\ s$

5 Memory Optimization

In the following, we describe a failed verification attempt to demonstrate the type of problems which can be detected with our approach.

In Solidity, access to storage variables is far more expensive than access to memory variables. Thus, instead of directly working on a storage variable, a common pattern is to first copy its content to memory, manipulate the corresponding memory variable, and finally copy the results back to storage. We can capture this pattern in another optimizer program which automatically replaces storage variables with corresponding memory variables. To this end, we first create three functions to update identifiers in $\mathcal{L}$-values, expressions, and statements, respectively. The corresponding function for $\mathcal{L}$-values, for example, looks as follows:

$$\mathit{lupdate}\colon \mathbf{S} \times \mathbf{S} \times \mathbf{L} \to \mathbf{L}$$

$$\mathit{lupdate}(j, j', \mathit{Id}(i)) = \begin{cases} \mathit{Id}(j') & \text{if } i = j \\ \mathit{Id}(i) & \text{if } i \neq j \end{cases}$$

$$\mathit{lupdate}(j, j', \mathit{Ref}(i, \mathit{xs})) = \begin{cases} \mathit{Ref}(j', \mathit{map}(\mathit{eupdate}(j, j'), \mathit{xs})) & \text{if } i = j \\ \mathit{Ref}(i, \mathit{map}(\mathit{eupdate}(j, j'), \mathit{xs})) & \text{if } i \neq j \end{cases}$$

where *map* is a higher-order function which executes another function over a sequence of values. The functions for expressions and statements are straightforward and thus not discussed further.

We can now define a function which implements the pattern discussed above:

$$\mathit{optimize}\colon \mathbf{S} \times \mathbf{S} \times \mathbf{MTypes} \times \mathbf{C} \to \mathbf{C}$$

$$\mathit{optimize}(v_s, v_m, \mathit{MTValue}(t), s) = \mathit{Block}\ ((v_m,\ \mathit{Value}(t), L(\mathit{Id}(v_s))), \mathit{up}(v_s, v_m, s))$$

$$\mathit{optimize}(v_s, v_m, \mathit{MTArray}(i, t), s) =$$
$$\mathit{Block}\ ((v_m, \mathit{Memory}(\mathit{MTArray}(i, t)), L(\mathit{Id}(v_s))), \mathit{up}(v_s, v_m, s))$$

where $\mathit{up}(v_s, v_m, s) \ = \ \mathit{supdate}(v_s, v_m, s)\ \mathbf{;}\ \mathit{Id}(v_s) \coloneqq L(\mathit{Id}(v_m))$

As an example, consider the following contract:

```
1  contract MyContract {
2    bool[1] sa;
3
4    function myFunction() public {
5      bool[1] memory ma = [false];
6      sa = ma;
7      sa[uint8(0)] = true;
8    }
9  }
```

```
bool[1] memory x = sa;
{
    x = ma;
    x[0] = true;
    sa = x;
}
```

Applying function *optimize* on it would replace the lines 6 and 7 with the program shown in the connected box. Again, it is important to ensure that *optimize* does not modify the semantics of programs and again we can formulate a corresponding correctness criterion in Isabelle.

To formulate the correctness statement, we first need to add two additional functions:

- Function $\mathit{fresh}(i, c)$ checks if an identifier i is not present in a statement c so far.
- Function $\mathit{convert}(t)$ converts a memory type to a corresponding storage type.

We can now define correctness of the optimizer program as follows:

$$\begin{aligned}&\mathit{fresh}(v_m, c) \ \wedge\ v_m \neq v_s \ \wedge\ \mathit{accessEnv}(v_s, e) = (\mathit{Storage}(\mathit{convert}(t_m)), v)\\ &\implies \mathcal{C}[\![c]\!]e\ s = \mathcal{C}[\![\mathit{optimize}(v_m, v_s, t_m, c)]\!]e\ s\end{aligned}$$

where $\mathit{accessEnv}(v, e)$ is used to obtain the type and value of a variable v in an environment e.

This time, when trying to verify the statement in Isabelle, it turns out that the statement does not hold in general. In particular, the substitution of reference type variables is critical. Consider, for example, again contract `MyContract` above. In the original program, line 6 copies the complete content of memory array ma to storage array sa. In line 7, the program then updates the value of the storage array without modifying ma. Indeed, given a definition of a corresponding environment env and state st, we can easily verify the following lemma in Isabelle:

Lemma 1. $\mathcal{C}[\![P]\!]e\ s = s' \ \wedge\ \mathit{access}(\text{“0.1”}, \mathit{mem}(s')) = \mathit{MValue}(\text{“}\mathit{False}\text{”})$

where P is the program consisting of lines 6 and 7 of contract `MyContract` and “0.1” is the location of the first element of array ma in memory.

On the other hand, the modified version of the program behaves as follows: First, it copies the complete content of storage array sa to the newly created memory array x. Now, however, since x is also a memory array, the semantics of the assignment $x = ma$ is different from the one in line 6 of the original program. Instead of copying again the content of the array, this time, the assignment just copies a pointer to the content of array ma to x. Therefore, the next line $x[0]$ = `true` does not only change the value of $x[0]$, but in addition it also changes the value of $ma[0]$. Thus, while the value of array sa after execution is the same for both programs, the optimized program has the additional side effect of changing also the content of array ma. Indeed, we can easily show the following lemma in Isabelle:

Lemma 2. $\mathcal{C}[\![\mathit{optimize}(sa, x, \mathit{MTArray}(1, \mathit{MTValue}(\mathit{TBool})), P)]\!]e\ s = s'$
$\wedge\ \mathit{access}(\text{“0.1”}, \mathit{mem}(s')) = (\mathit{MValue}(\text{“}\mathit{True}\text{”}))$

6 Related Work

Early work on formalizing Ethereum smart contracts has focused on the Ethereum Virtual Machine (EVM) [40]. One of the first examples in this area is the work of Hirai [20], which provides a formalization of the EVM in Lem [27].

Later on, Hildebrandt et al. provide an alternative formalization using the $\mathbb{K}$-framework [33] called KEVM [19]. Around the same time, Grischenko et al. [16] provide a formalization of the EVM in F* [36] and Amani et al. one for the interactive theorem prover Isabelle/HOL [4]. All the work in this area describes the formalization of the Ethereum Virtual Machine to support the verification of contracts at the byte-code level. With our work we focus on the higher level language Solidity which allows more abstract reasoning.

More recently, also work on formalizing and analyzing smart contracts in Solidity emerged: Bhargavan et al. [9], for example, describe an approach to map a Solidity contract to F* where it can then be verified. In addition, Mavridou et al. [26], provide an approach based on FSolidM [25], in which a Solidity smart contract is modeled as a state machine to support model checking of common security properties. TinySol [6] and Featherweight Solidity[12], on the other hand, are two calculi formalizing some core features of Solidity. Crosara et al. [13] describe an operational semantics for a subset of Solidity. Moreover, Ahrendt and Bubel describe SolidiKeY [3], a formalization of a subset of Solidity in the KeY tool [2] to verify data integrity for smart contracts. In addition, Zakrzewski [42] 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].

While all these works provide important insights into the formal foundation of Solidity, most of them are not executable and therefore difficult to evaluate. On the other hand, we considered it important to have an executable semantics that can be evaluated against the reference implementation. We share this desire with Hajdu and Jovanovic [17,18], which provide a formalization of Solidity in terms of a simple SMT-based intermediate language which they evaluate on a set of manually developed tests. In addition, Jiao et al. [22,23], provide a formalization of Solidity in $\mathbb{K}$ with a rigorous evaluation using the Solidity compiler test set.Our work differs from the above works mainly in two main aspects:

1. We provide the first implementation of a semantics for Solidity for the interactive theorem prover Isabelle/HOL.
2. Our approach comes with an integrated fuzzy-testing framework which allows to automatically test the semantics against the reference implementation every time the semantics is updated.

7 Conclusion

We presented a formal semantics of Solidity, as a conservative extension of Isabelle/HOL. Our work includes a test framework for automatically validating that our semantics describes the behavior of the actual Solidity implementation faithfully. As a first step of showing the usefulness of our semantics, we demonstrated the formal analysis of two different optimizations of Solidity programs that potentially help to make smart contracts more "gas efficient".

In our current work, we focused on the core of the Solidity language and the more exotic features such as its memory model and the numerous types of integers. We plan to extend the formalization with support for missing language features such as function calls. And we also plan to improve and extend the verification framework, e.g., by providing support for the keywords require and assert, and a verified verification condition generator. Moreover, we started already to increase the level of proof automation by developing domain specific tactics.

Availability. Our formalisation, the test framework, and the evaluation results are available under BSD license (SPDX-License-Identifier: BSD-2-Clause) [24].

Acknowledgements. We would like to thank Tobias Nikpkow for useful discussions about the compliance testing. Moreover, we would like to thank Silvio Degenhardt and Nick Papavasileiou for their support with implementing the semantics.

References

1. The Bitcon market capitalisation., https://coinmarketcap.com/currencies/bitcoin/, last checked on 2021-05-04.
2. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive software verification–the KeY book, vol. 10001. Springer (2016)
3. 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 (2020)
4. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: CPP. p. 66–77. CPP 2018, ACM (2018). https://doi.org/10.1145/3167084
5. Ballarin, C.: Interpretation of locales in isabelle: Theories and proof contexts. In: Borwein, J.M., Farmer, W.M. (eds.) Mathematical Knowledge Management. LNCS, vol. 4108, pp. 31–43. Springer (2006). https://doi.org/10.1007/11812289_4
6. 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)
7. 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)
8. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT. pp. 313–314. Springer (2013)
9. 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
10. ConsenSys Software Inc.: Ganache. https://www.trufflesuite.com/docs/ganache/, Accessed: 2021-05-01
11. ConsenSys Software Inc.: Truffle. https://www.trufflesuite.com/truffle, Accessed: 2021-05-01

12. 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)
13. 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)
14. Gill, A., Runciman, C.: Haskell program coverage. In: Haskell Workshop. p. 1–12. Haskell '07, ACM (2007). https://doi.org/10.1145/1291201.1291203
15. Gordon, M.: From LCF to HOL: a short history. In: Plotkin, G., Stirling, C., Tofte, M. (eds.) Proof, Language, and Interaction: Essays in Honour of Robin Milner, pp. 169–185 (2000)
16. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) Principles of Security and Trust. pp. 243–269. Springer (2018)
17. 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
18. 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
19. Hildenbrandt, E., Saxena, M., Rodrigues, N., Zhu, X., Daian, P., Guth, D., Moore, B., Park, D., Zhang, Y., Stefanescu, A., Rosu, G.: Kevm: A complete formal semantics of the Ethereum virtual machine. In: CSF. pp. 204–217 (2018). https://doi.org/10.1109/CSF.2018.00022
20. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: Brenner, M., Rohloff, K., Bonneau, J., Miller, A., Ryan, P.Y., Teague, V., Bracciali, A., Sala, M., Pintore, F., Jakobsson, M. (eds.) Financial Cryptography and Data Security. pp. 520–535. Springer (2017)
21. Hodován, R., Kiss, A., Gyimóthy, T.: Grammarinator: A Grammar-Based Open Source Fuzzer. In: Automating TEST Case Design. p. 45–48. A-TEST 2018, ACM (2018). https://doi.org/10.1145/3278186.3278193
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. Marmsoler, D., Brucker, A.D.: A denotational semantics of Solidity in Isabelle/HOL: Implementation and test data (Oct 2021). https://doi.org/10.5281/zenodo.5573225
25. 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)
26. Mavridou, A., Laszka, A., Stachtiari, E., Dubey, A.: Verisolid: Correct-by-design smart contracts for Ethereum. In: Goldberg, I., Moore, T. (eds.) Financial Cryptography and Data Security. pp. 446–465. Springer (2019)
27. Mulligan, D.P., Owens, S., Gray, K.E., Ridge, T., Sewell, P.: Lem: Reusable engineering of real-world semantics. SIGPLAN Not. **49**(9), 175–188 (Aug 2014). https://doi.org/10.1145/2692915.2628143
28. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)

29. Online: Remix – Solidity IDE. https://remix-ide.readthedocs.io/en/latest, Accessed: 2021-05-01
30. Online: Solidity documentation. https://docs.soliditylang.org/en/v0.5.16/, Accessed: 2021-05-01
31. Parr, T.: Antlr (another tool for language recognition). https://www.antlr.org/index.html, Accessed: 2021-05-01
32. Perez, D., Livshits, B.: Smart contract vulnerabilities: Vulnerable does not imply exploited. In: USENIX Security. USENIX Association (Aug 2021)
33. Roșu, G., Șerbănuță, T.F.: An overview of the K semantic framework. The Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). https://doi.org/10.1016/j.jlap.2010.03.012
34. Scott, D.: Outline of a mathematical theory of computation. Oxford University Computing Laboratory, Programming Research Group Oxford (1970)
35. Scott, D., Strachey, C.: Toward a mathematical semantics for computer languages, vol. 1. Oxford University Computing Laboratory, Programming Research Group Oxford (1971)
36. Swamy, N., Hriţcu, C., Keller, C., Rastogi, A., Delignat-Lavaud, A., Forest, S., Bhargavan, K., Fournet, C., Strub, P.Y., Kohlweiss, M., Zinzindohoue, J.K., Zanella-Béguelin, S.: Dependent types and multi-monadic effects in F*. In: Symposium on Principles of Programming Languages. p. 256–270. POPL '16, ACM (2016). https://doi.org/10.1145/2837614.2837655
37. The Coq development team: The Coq proof assistant reference manual. LogiCal Project (2004), version 8.0
38. Wenzel, M.: Isabelle/Isar – a generic framework for human-readable proof documents. From Insight to Proof – Festschrift in Honour of Andrzej Trybulec **10**(23), 277–298 (2007)
39. Wood, G.: Ethereum: A secure decentralised generalised transation ledger (version 2021-04-21). Tech. rep.
40. Wood, G., et al.: Ethereum: A secure decentralised generalised transaction ledger. Ethereum project yellow paper **151**(2014), 1–32 (2014)
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. 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