Towards verification of Ethereum smart contracts: a formalization of core of Solidity

Jakub Zakrzewski[⋆]

University of Warsaw
ul. S. Banacha 2a, 02–097 Warsaw, Poland
j.zakrzewski@mimuw.edu.pl

Abstract. Solidity is the most popular programming language for writing smart contracts on the Ethereum platform. Given that smart contracts often manage large amounts of valuable digital assets, considerable interest has arisen in formal verification of Solidity code. Designing verification tools requires good understanding of language semantics. Acquiring such an understanding in case of Solidity is difficult as the language lacks even an informal specification.
In this work, we evaluate the feasibility of formalization of Solidity and propose a formalization of a small subset of Solidity that contains its core data model and some unique features, such as function modifiers.

Keywords: Solidity · Ethereum · Smart contracts · Semantics.

1 Introduction

Ethereum is a blockchain-based platform that provides a globally-consistent virtual general-purpose computer, called the Ethereum Virtual Machine. The programs to be executed on the EVM, called *smart contracts*, are provided in a stack-based machine language, which has a corresponding assembly language. But most smart contracts are written in higher-level languages. The most popular language of those is called Solidity. Given that Ethereum smart contracts often manage assets worth millions of US dollars, bugs in their design may lead to enormous harm. Since there is so little margin for error, considerable interest has arisen in formal verification of Solidity code.

To design accurate verification tools, one needs to know precisely what Solidity is. Superficially, Solidity seems to be a simple language that should be easy to understand for anyone familiar with mainstream programming languages of the C family, as it was designed with similarity to JavaScript in mind. However, after reading the freely available documentation while investigating the possibility of creating such a tool, we realized that we do not really understand the language well. In the pursuit of providing high-level features, while maintaining the abstractions efficiently implementable for the Ethereum Virtual Machine, the creators of the language ended up with a language that has a very different type system and

[⋆] This work was partially supported by the Polish NCN grant 2013/11/B/ST6/01381.

data model, and numerous unusual features. To improve our understanding of Solidity, we needed a precise specification.

Although much work has gone into formalizing and verifying the Ethereum Virtual Machine bytecode [9, 8, 12, 1, 7], there appears to be a paucity of studies on Solidity. A previous work on the topic by Bhargavan et al. [5] does not give explicit semantics. As such, we set out to create a new formalization. Since even an informal specification of the Solidity language does not exist, we had to design the semantics by studying the official documentation, doing experiments, and analysis of Solidity compiler sources. To enable establishing trust in the specification, we implemented it in an executable form in Coq, in a way such that it is possible to extract a working interpreter of the language. In the future, this will enable testing the semantics and comparing it with the real implementation.

Semantics for the complete Solidity language will be rather complex due to the multitude of features the language provides. In this paper:

- We give an overview of Solidity and highlight some surprising and, in our opinion, poorly documented features of the language, like modifiers and the data model (Section 2). We consider providing an accurate description of these language features to be a necessary step towards creating sound and complete verification tools for Solidity.
- We describe the dynamic semantics of several core Solidity constructs formally. Our semantics is given in an executable form in Coq, however here, it is described in conventional metanotation (Section 3). Our semantics omit many features we consider inessential to enabling deductive verification of contract state invariants. For example, we do not model transaction fees. Since providing insufficient funds for a transaction fee simply causes the whole transaction to abort without affecting the contract state.

2 Overview of Solidity

Solidity was originally designed as a JavaScript-like, but typed, programming language for the Ethereum platform. The language is structured into contract definitions, function and modifier definitions, statements, and expressions. A Solidity source unit is composed of contract definitions. Fig. 1 presents the abstract syntax of the described fragment of Solidity.

2.1 Contracts and messages

At source level, contracts appear similar to classes in object-oriented languages. They can contain declarations of *state variables* (analogous to class fields), definitions of functions (method), modifiers, constructors, and structs (record type), they support encapsulation (visibility attributes), and can even inherit from multiple other contracts. All the defined entities reside in a single namespace. Functions can be statically overloaded. For resolving the visibility of inherited identifiers the creators of Solidity opted for the widely used C3 linearization algorithm [4].

e	::=	*var*	local variable access
	|	*var{cname}*	state variable access
	|	*funname*	internal function
	|	*cst*	`bool` or `uint` constants
	|	e_1 `(` e_2* `)`	function call
	|	e_1`[`e_2`]`	array access
	|	*e* `.` *fname*	member access
	|	*op e*	unary arithmetic or logic operation
	|	*e* ≪ *op* ≫ *e*	binary arithmetic or logic operation
	|	`this`	
	|	e_1 `=` e_2	assignment
	|	*e**	tuple
cst	::=	`true` | `false`	Boolean constant
	|	*n*	integer constant
StorageLocation	::=	`memory` | `storage`	
LocalVarDef	::=	*TyName StorageLocation? var*	local variable declaration
TyName	::=	`uint`	unsigned machine integer
	|	`bool`	boolean
	|	*sname*	struct
	|	*TyName*`[]`	dynamically sized array
	|	*TyName*`[`*n*`]`	fixed size array
	|	`mapping(`*TyName* `=>` *TyName*`)`	mapping
stmt	::=	$stmt_1$`;` $stmt_2$	sequence
	|	`if` *e* `then` *stmt* `else` *stmt*	conditional
	|	`while (`*e* `)` *stmt*	while loop
	|	`throw;`	
	|	`return;`	
	|	`return` *e*`;`	return statement
	|	`_;`	placeholder
	|	*LocalVarDef* `;`	local variable declaration
StructFieldDef	::=	*TyName id*	
StructDef	::=	`struct` *sname* `{` *StructFieldDef** `}`	

FunDef	::=	`function` [*funname*]`(`*LocalVarDef**`)`
		[*mname* `(` *e** `)`]*
		[`returns (` [*TyName* [*var*]]* `)`]
		`{` *stmt* `}`
			function definition

ModifierDef	::=	`modifier` *mname*(*id**) `{` *stmt* `}`	modifier definition
StateVarDef	::=	*TyName StorageLocation id*	
ContractDef	::=	`contract` *cname* `is` *cname** { *StateVarDef* FunDef** *ModifierDef* StructDef** }	contract definition

Fig. 1. Abstract syntax of our core Solidity.

To enable deployment on the Ethereum platform, the contract functions are compiled into EVM bytecode and a piece of code called *function selector* is added, which serves as an entry point into the contract code. The resulting EVM program is put in a *contract account* on the Ethereum network, which, in addition to storing code, also holds a certain amount of virtual currency. From that point on, other Ethereum accounts may trigger execution of contract functions or currency transfers by sending *messages* to the contract account. Whenever someone sends a message to the contract account, the contract code starts executing at the function selector. The selector decodes the message and jumps into an appropriate contract function (method). Solidity supports custom handling of messages that do not specify a concrete function to call, by the way of specifying *fallback functions*.

Functions return values using *return variables*. Their names can be either explicitly specified by the programmer, or it may be unique identifiers generated during the typechecking phase.

The declarations of local variables, in addition to type names and variable names can have a *storage location*. Local variables of simple scalar types, i.e. machine integers and Ethereum network addresses, do not need, or allow, this annotation. They always reside on the EVM stack. However, in addition to those simple types, we can declare *pointers* to compound types, such as structs and arrays. Objects of these types can reside in two locations: memory and storage. These are described in more detail in following subsections.

State variables State variables can store *storage objects*. Storage objects can be either scalars (machine integers, addresses, booleans), arrays of storage objects (either fixed-size or resizable), structs (records of storage objects) or mappings. Mappings are a sort of hash tables that are able to map certain hashable types to storage objects. However unlike typical hash tables, they do not store the keys alongside the values and do not implement any collision resolution mechanism. Instead, a key hashing scheme based on a collision resistant cryptographic hash function is used to derive an address where the value is stored [19] in the large 256-bit virtual address space of EVM storage. This lack of collision resolution complicates formal reasoning about operations on mappings, so we decided to treat this aspect as an implementation detail and abstract it away in our formalization, i.e. treat mappings as formal mappings, with no hashing involved.

The state variables reside in persistent *storage* of a given account, i.e. they are a part of the global state of the Ethereum network.

Note that the storage, as viewed from Solidity, does not behave like a traditional automatically managed heap of objects linked by pointers. Instead, storage objects are treated as values. As a result, assigning directly to a state variable causes deep copying, as demonstrated in Fig. 2. Notably though, this does not work for mappings, as they cannot be copied. Still, structs with a field of mapping type can be copied, though the problematic field in the destination object is simply left alone, preserving the old values.

```
contract Z  {
  int[][] baz;
  function foo() returns(int) {
    baz.length = 10; // resize baz
    baz[0].length = 10;  // resize baz[0]
    baz[1].length = 1;  // resize baz[1]
    int[] storage bar = baz[0];  // bar is a storage pointer
    baz[0] = baz[1];  // deep copy
    return bar[9];   // error
  }
}
```

Fig. 2. Direct storage assignment example.

That said, pointers to non-scalar storage objects can be taken and stored in local variables. An important thing to notice is that storage objects have automatic lifetimes, tied to reachability from a state variable. For example, calling the function `foo` of the following contract in Fig. 3 results in an error.

```
contract Z  {
  int[][] baz;
  function foo() returns(int) {
    baz.length = 10; // resize baz
    baz[0].length = 10;
    int[] storage bar = baz[0];  // bar is a storage pointer
    baz.length = 0;  // bar becomes a dangling pointer!
    return bar[0];   // error
  }
}
```

Fig. 3. An example demonstrating storage data structure lifetime.

Modifiers A unique feature of Solidity are the function modifiers, a functionality somewhat similar to Python decorators. They are often used to add precondition checks to contract functions. The modifiers assigned to a function are executed before entering the actual function body. The modifier hands over the control flow to the next modifier or the function body when so-called placeholder statement is encountered (`_;`).

```
  address owner;
  modifier my_modifier(address a) {
    if (a != owner) { throw; }
    _; // enter the function body
```

```
}
function foo() my_modifier(msg.sender) returns (uint) {
  uint a; a += 1;
  return a;
}
```

The official Solidity documentation does not go into much detail on the semantics of modifiers. It only gives examples of the very simplest cases. For example, consider code in Fig. 4.

```
modifier my_modifier1 {
  _; _; _; // enter the function body three times
}
modifier my_modifier2 {
  uint a;
  if (a > 1) { _; }
  a += 1; _;
}
modifier my_modifier3(uint a) {
  if (a < 2) { _; }
}
function foo(uint a) my_modifier1 my_modifier2 my_modifier3
    (a) returns (uint) {
  a += 1; return a;
}
function bar() returns (uint) {
  return foo(0); // returns 2
}
```

Fig. 4. Example of confusing modifier code.

Multiple things are not obvious about the behavior of the function foo() and are not described by the documentation:

- Are function's local variable values preserved when it is entered multiple times from modifiers? Our experiments indicate that they are;
- Are modifier's local variable values preserved when it is entered multiple times from other modifiers? In this case it seems that they are not;
- When are arguments passed to modifiers evaluated? They are evaluated anew each time the modifier is entered.

Struct definitions A struct definition in Solidity consists of a list of pairs of type names and field names. The definition can be used to instantiate both memory and storage objects, and the storage location of the object is propagated to fields. This means e.g. that pointers to storage objects cannot be stored in memory structs.

2.2 Type names

The internal type system of the Solidity compiler is richer than the syntax suggests, as it tracks additional type attributes, like storage location for composite types. As a result, we have to differentiate type names from the actual types. The actual type of the declared entity depends on the context of the declaration and annotations provided by the programmer. This may be directly observed in error messages produced by the compiler. For example, a local variable declared as `uint[] storage a` has type `uint[] storage pointer`, while a storage variables declared as `uint[] b` has type `uint[] storage ref`.

2.3 Statements

We consider a limited set of simple control structures, such as if/then/else, while loops. Additional features such as for and do/while loops are left out, as their semantics are not particularly unusual.

One novel construct found in Solidity is the placeholder statement `_;`. It may appear only in function modifiers, where it denotes the point of entry into the next modifier or the function body.

The `return` statements interrupt the control flow and jump out of the function body. Providing an explicit value to the `return` statement causes it to have the side effect of assigning to the return variable. The control flow returns either to the caller or to a function modifier if one is used. The modifier may then reenter the function body. In that case all values of local variables, including return variables, are preserved.

2.4 Expressions

The annotation *cname* in state variable access and assignment statements of our abstract language denotes the contract (class) where the referenced variable is defined. It is not a part of Solidity's concrete syntax and is inferred by the type checker.

Function calls in Solidity can be of several types: internal, external, delegate, and calls to certain builtin functions. Internal function calls are simply jumps in the code of the current account. External calls cause a message to be sent over the Ethereum network, executing code on another account. Delegate calls exist to provide a functionality akin to shared libraries. That is, they allow code from another account to directly operate on the storage of calling account. The semantics of external and delegate calls are notorious source of bugs in contracts [2], notably the DAO [6] and Parity multi-sig contract [16] incidents. Since the code executed by outgoing external function calls may not be available, or not written in Solidity, we decided to specify the behavior of such calls only in terms of axioms that effectively state that arbitrary changes to the network state could be made. This is not very helpful for verification purposes, however a provision could be made for preservation of certain global invariants.

The semantics of this keyword in Solidity is quite unusual. In Solidity this is the address of the current account in the Ethereum network, which the contract can use to send an Ethereum message to itself. Directly accessing state variables using this is not possible, and though accessor functions are generated for public state variables, calling them incurs the cost of a message call. These phenomena are demonstrated by Fig. 5.

```
contract Foo {
  uint private i;
  uint public j;
  function bar() returns (uint) {
    // return this.i;     // this would cause an error
    return this.j();      // externally calls an
        automatically generated getter
  }
}
```

Fig. 5. this cannot be used to access state variables directly.

An especially unusual corner case is using this in constructors. The constructors execute *before* this account is actually able to receive and decode messages in the Ethereum network, which means that dispatching calls on this in a constructor, as in Fig. 6, causes a runtime error.

```
contract Foo {
  constructor() { this.foo(); }  // this causes an error!
  function foo() { }
}
```

Fig. 6. this incorrectly used in a constructor.

The order of evaluation of sub-expression in Solidity is explicitly left unspecified. Since this is tangential to the concepts we want to explore in this paper, we assume for simplicity a deterministic order.

Interestingly, the Solidity language has no constants of machine integer types. Instead, all numeric constants in the source are treated by the compiler as arbitrary precision rational numbers. At the typechecking stage, the constants are folded, and the results are converted to machine integers of type appropriate to the source context. However, since this step can be done entirely statically, we do not model this in our semantics.

2.5 Memory objects in Solidity

As mentioned before, Solidity programs have access to auxiliary volatile memory. The view of memory as provided by Solidity is that of a mapping of pointers to memory objects. Memory objects can be either arrays or structs, which can contain scalars or pointers to other memory objects.

Note that there are no mappings in memory. Also, unlike in the case of storage, memory objects cannot be contained in other memory objects as values. Importantly, this means that a single struct declaration in Solidity can have two very different concrete representations.

```
struct S {
  int[8] a;
  mapping (int => int) m;
}
```

Fig. 7. An example struct declaration.

For example, when the struct definition in Fig. 7 is instantiated in memory, we get an object that contains a single field `a` that is a pointer to a memory array. An attempt to access `m` results in a type error. On the other hand, a state variable of type `S`, contains an object that has two fields, one of them being a storage array object (not a pointer), and the other a mapping.

Somewhat confusingly though, whenever a local variable of memory pointer type is defined, it is automatically initialized to point to a newly allocated memory object of given type and this initialization is recursive, i.e. all nested pointers in the object are initialized the same way. In fact, Solidity does not allow the programmer to explicitly allocate memory objects other than dynamically sized arrays, nor it provides any facility to free allocated objects or otherwise reclaim allocated memory. Thus, the code in Fig. 8, unlike the similar code in Fig. 2, does not result in an error being raised.

3 Formalization

Our formalization of Solidity focuses on dynamic semantics and it is written as an interpreter in Coq, in monadic functional big-step semantics style described by Owens et al. [15], however in this paper it is presented in a more conventional notation. Describing the semantics of the full Solidity language, including its type system, is too big to fit in a workshop paper. In order to make our presentation feasible, we focus here only on a subset of Solidity that captures the following features:

- contracts with storage,
- memory,

```
contract Z  {
  function foo() returns (int) {
    int[][] memory baz = new int[][](10);
    baz[0] = new int[](10);
    baz[1] = new int[](1);
    int[] memory bar = baz[0];
    baz[0] = baz[1];
    return bar[9]; // OK
  }
}
```

Fig. 8. Memory allocation.

- inheritance,
- modifiers.

We omit other features, such as function overloading, visibility specifiers, rational constant types, integers of sizes less than 256 bits, packed byte array types, libraries, events, and most of Solidity's global built-in functions and variables. We assume all functions are callable both internally and externally. These features were left out, as they are either laborious to implement or not very interesting from the point of view of dynamic semantics. We have aimed for the formalization to be as abstract as possible, for example by not exposing the EVM data model, to make reasoning about programs simpler.

The Ethereum network The main effect of executing smart contract code is altering the state of the Ethereum network. From the point of view of our formalization, the Ethereum network σ is a partial mapping of addresses to accounts.

An account is a triple $\langle b, p, s\rangle$ of balance b (the amount of currency the account holds), the contract program p, and storage s. The contract program p can be thought of as a pair $\langle c, \mathit{cdefs}\rangle$ of contract name and a list of definitions of the account contract, as well as all its parents in the inheritance hierarchy. To model the object-oriented nature of Solidity contracts, we abstract from the low-level EVM view of the account storage as a mapping of machine words to machine words, and instead we consider it to be analogous to a field table in Jinja [14]. Therefore, a storage s is a partial mapping from pairs $(\mathit{var}, \mathit{cname})$ to storage objects so, where var is the name of the variable and cname is the identifier of the contract (class) where the variable was defined. Both those components are needed, since inheritance may lead to a single contract containing more than one variable of the same name. Storage objects can be thought of as trees containing values in their leaves and with mappings, arrays or structs as their nodes:

$$\begin{aligned}
\mathit{so} ::=\ & \texttt{Smapping}\ (m),\ \text{where } m \in v \to \mathit{so} \\
\mid\ & \texttt{Sarray}\ (a, \mathit{typ}, l),\ a \in \mathbb{Z} \to \texttt{option}(\mathit{so}), l \in \mathbb{Z} \\
\mid\ & \texttt{Sstruct}\ (s),\ s \in \texttt{ident} \to \texttt{option}(\mathit{so}) \\
\mid\ & \texttt{Sval}\ (v)
\end{aligned}$$

This is similar to the model of C++ objects proposed by Ramananandro et al. [17]. Solidity mappings are total, initially mapping all keys to default objects of declared value type. Arrays contain, aside from partial mappings from indices to storage objects, length, and type information to allow bounds checking and initializing objects in new cells when resizing.

The contract execution is done as a part of a *transaction* which can be triggered by *messages* sent to the network. Two kinds of messages are currently of interest to us: creation messages, which are used to create new contract accounts in the network, and normal call messages. We do not attempt to formalize the Solidity ABI specification. Instead, for our purposes, creation messages are quadruples $\langle a_s, v, p, vs\rangle$ and normal messages are quintuples $\langle a_s, a_r, v, \mathit{funname}, vs\rangle$, where a_s and a_r are the network addresses of the sender and recipient accounts, the *value* v that holds the amount of currency sent, p is contract code, *funname* is a name of a function to call, and vs is a list of values, which are the arguments to the constructor or the called function.

Memory Memory m can be thought of as a mapping $loc \to mo$ from a set of locations to memory objects. Memory objects can be either arrays or structs that store values, including pointers to other memory objects:

$$\begin{array}{ll} mo ::= & \texttt{Marray}(a),\ a \in \mathbb{Z} \to \texttt{option}(v) \\ & \mid \texttt{Mstruct}(s),\ s \in \texttt{ident} \to \texttt{option}(v) \end{array}$$

Since memory arrays are not resizable, unlike storage arrays, they do not need to carry type information. Storage references in memory are disallowed by the type system. We use m_{empty} to denote an empty memory.

Values Values range over 256-bit machine integers, Booleans, storage references, memory pointers, and internal or external function pointers.

$$\begin{array}{lll} v & ::= \texttt{Vbool}\ (b),\ b \in \{\text{true}|\text{false}\} & \text{booleans} \\ & \mid \texttt{Vint}\ (n),\ n \in \mathbb{Z}, 0 \le n < 2^{256} & \text{machine integers} \\ & \mid \texttt{Vaddr}\ (a),\ a \in \mathbb{Z}, 0 \le n < 2^{160} & \text{addresses} \\ & \mid \texttt{Vsref}\ (\mathit{sref}) & \text{storage references} \\ & \mid \texttt{Vmptr}\ (\mathit{loc}) & \text{memory pointers} \\ & \mid \texttt{Vifptr}\ (\mathit{funname}, \mathit{cname}) & \text{internal function pointers} \\ & \mid \texttt{Vefptr}\ (a, n, \mathit{funname}) & \text{external function pointers} \\ \mathit{sref} & ::= \texttt{SRmapping\_val}(\mathit{sref}, v) & \\ & \mid \texttt{SRarray\_cell}(\mathit{sref}, i),\ i \in Z & \\ & \mid \texttt{SRsfield}(\mathit{sref}, \mathit{funname}) & \\ & \mid \texttt{SRvar}(\mathit{var}, \mathit{cname}) & \end{array}$$

Storage references are paths to (sub)objects in the storage of the current account. This accurately models the reachability and lifetimes of storage objects. Memory pointers, on the other hand, are simply locations. Internal function pointers contain a function name and the identifier of the contract where it is defined. External function pointers contain an address of an external account, value (i.e. the amount of currency to send along with the call), as well as a function name.

Lvalues Lvalues are entities that designate an assignable location in one of the available storage locations. Lvalues may either point to locals, storage objects, cells in memory arrays, fields in memory structs, or tuples.

$lv ::=$ `LVlocal` (var)	local variable
$\mid$ `LVstorage` $(sref)$	storage reference
$\mid$ `LVmem_arr_cell` (loc, i), $i \in \mathbb{Z}$	memory array cell
$\mid$ `LVmem_sfield` $(loc, fname)$	memory struct field
$\mid$ `LVtuple` $(lv*)$	tuple

Most of these are self-explanatory. Tuple lvalues are a syntactic construct mostly used to retrieve multiple return values from a function. They are produced by tuple expressions appearing in an lvalue context. Tuple values do not exist, so they cannot be assigned to variables or passed as parameters.

3.1 Big step semantics

State State μ is a tuple $\langle a, m, \sigma, l^f, l^m \rangle$ of the address a of the current account, memory m, network σ, function local store l^f, and modifier local store l^m. To accurately model the semantics of Solidity function modifiers, we introduce two stores for local variables: a function local store and a modifier local store. These stores are partial mappings from identifiers to values. A function store is used to hold local variables of the currently executing function, while a modifier store is used to hold local variables of currently executing modifier, if any. We use l_{empty} to denote an empty store.

Evaluation The evaluation judgments are of the following forms:

$\vdash msg, \sigma \Rightarrow \sigma'$	transactions
$p, q, f \vdash stmt, \mu \Rightarrow out, \mu'$	statements
$p \vdash e, \mu \Rightarrow out, \mu'$	expressions
$p \vdash e, \mu \Leftarrow out, \mu'$	expressions in lvalue position

The entry point to our semantics is the transaction judgment. A transaction judgment relates a message msg and a network state σ to a new network state σ'. The judgments for statements, expressions, and lvalues relate corresponding syntactic elements to the outcomes of their execution in the given state μ. The symbol q denotes the *modifier stack*, while f is the function being executed. The modifier stack stores pairs $\langle fm, e* \rangle$ of function modifiers that are yet to be executed and lists of their unevaluated arguments (i.e. expressions). The top of the stack contains the modifier that is entered when the next placeholder statement is encountered. The return variables are used to store the return values of the currently executing function.

The rules for expressions and statements may produce one of the following outcomes:

$out ::=$ `OK`(v) $\mid$ `OK`(lv) $\mid$ `OK`(vs) $\mid$ `Return` $\mid$ `Fail`

The `OK` outcome means normal termination. A successful termination may yield a value v, lvalue lv, or a list of values vs. The `OK` notation is "overloaded" respectively. `Return` is used to interrupt control flow and jump out of the function body upon encountering a `return` statement. The `Fail` outcome is used to propagate exceptions, which cannot be caught and always cause the transaction to fail. In Solidity execution may fail for several reasons, like performing invalid storage accesses or transaction fees exceeding allowances, however we are currently not interested in tracing the causes of these failures.

Space constraints make it impossible to exhaustively list all the rules of our core language, so we give only a few examples of rules we consider interesting.

Account creation. First things first, we give a rule describing contract account creation. Whenever someone wants to deploy a new contract on the Ethereum network, they have to send a creation message containing contract code. This results in a new account being created and the contract's constructor being run.

$$\frac{\begin{array}{c} msg = \langle a_s, v, p, vs \rangle \\ \sigma_1(a) = \texttt{None} \qquad\qquad \sigma_2 = \sigma_1[a \rightarrow \langle v, p_{empty}, mkstorage(p) \rangle] \\ \mu = \langle a, m_{empty}, \sigma_2, mklocals(c, vs), l_{empty} \rangle \\ run\_constructors(p, \mu) = \mu' \quad \mu' = \langle \ldots, \sigma_3, \ldots \rangle \quad \sigma_4 = \sigma_3[a \rightarrow \langle \ldots, p, \ldots \rangle] \end{array}}{\vdash msg, \sigma_1 \Rightarrow \epsilon, \sigma_4}$$

Here, the account with address a_s (the sender) sends a creation message into the network, with the aim of deploying the contract p. The rule for handling creation messages generates a fresh network address a. Under this address, a new account with empty contract code, denoted by p_{empty}, is stored, resulting in a modified network σ'. The $mkstorage(p)$ function creates a new storage, with all storage variables in p set to default values. Similarly, $mklocals(c, vs)$ creates a new local variable store with the arguments of c mapped to vs and locals initialized to default values. Then, constructors of the contract and its superclasses are run according to reverse MRO (the C3 algorithm [4]), i.e. from the most basic to the most derived. Only when all these steps execute successfully, the contract code is actually stored[1] in the network σ_4. The result of the transaction is an empty list of values ϵ and the network σ_4. Note that executing constructors does not require special handling of internal virtual function calls. Our experiments show that functions called are always those of the most derived contract, even if its constructor has not yet finished executing.

Handling normal messages. Once we have a contract account on the network, we may execute its functions. This is done by sending messages with the appropriate value of the recipient field, as captured by the following rule:

$$\frac{\begin{array}{c} msg = \langle a_s, a, v, fn, vs \rangle \qquad \sigma_1(a) = \langle b, p, s \rangle \qquad MRO(fn, p) = c \\ lookup(p, fn, c) = f \\ \mu_1 = \langle a, m_{empty}, \sigma_1[a \rightarrow \langle b + v, p, s \rangle], mklocals(f, vs), l_{empty} \rangle \\ p, modifiers(f, p), retvars(f) \vdash \_;, \mu_1 \Rightarrow \texttt{OK}(\epsilon), \mu_2 \quad \mu_2 = \langle \ldots, \sigma_2, l_2^f, \ldots \rangle \\ rvs = [l_2^f(retvar_1(f)), \ldots, l_2^f(retvar_n(f))] \end{array}}{\vdash msg, \sigma_1 \Rightarrow rvs, \sigma_2}$$

[1] Readers acquainted with the internals of Solidity might notice that, unlike the real thing, we do store the constructor code in the network. However, after the account is created, the constructors become inaccessible anyway.

where the function *retvars* simply extracts the return variables of a given function, while *modifiers* looks up modifier definitions and builds the modifier stack. Here, the sender a_s tries to trigger the execution of a function[2] called *fn*. This rule basically performs the role of the function selector. $\mathit{MRO}(\mathit{funname}, p)$ returns the identifier of the contract where according to the method resolution order the function should be looked up. The *lookup* function retrieves the function definition. Then we set the modifier stack to contain all the called function's modifiers and reuse the rule for the placeholder statement to actually start executing the first function modifier, or otherwise enter the function body. After the execution of the function terminates, the values of its return variables $\mathit{retvar}_1(f), \ldots, \mathit{retvar}_n(f)$ are extracted and returned. Note that we are assuming any function can receive currency, but in Solidity only those declared as *payable* can do that.

Internal function calls. The internal call of a function given as a pointer is similar, however not messages are involved:

$$\frac{\begin{array}{c} p \vdash e_1, \mu_1 \Rightarrow \texttt{OK}(\texttt{Vifptr}(fn, c)), \mu_2 \qquad p \vdash e_2{*}, \mu_2 [\Rightarrow] \texttt{OK}(vs), \mu_3 \\ \mathit{lookup}(p, fn, c) = f \qquad\qquad \mu_3 = \langle a, m_3, \sigma_3, l_3^f, l_3^m \rangle \\ \mu_4 = \langle a, m_3, \sigma_3, \mathit{mklocals}(f, vs{*}), l_{empty} \rangle \\ p, \mathit{modifiers}(f, p), f \vdash \_; , \mu_4 \Rightarrow \texttt{OK}(\epsilon), \mu_5 \\ \mu_5 = \langle \ldots, l_4^f, \ldots \rangle \qquad rvs = [l_4^f(\mathit{retvar}_1(f)), \ldots, l_4^f(\mathit{retvar}_n(f))] \end{array}}{p \vdash e_1(e_2{*}), \mu_1 \Rightarrow \texttt{OK}(rvs), \mu_5}$$

where $[\Rightarrow]$ means extension of evaluation to lists of expressions.

External function calls. External calls are specified in terms of an *external_call* axiom:

$$\frac{\begin{array}{c} p \vdash e_1, \mu_1 \Rightarrow \texttt{OK}(\texttt{Vefptr}(a, v, fn)), \mu_2 \qquad p \vdash e_2{*}, \mu_2 [\Rightarrow] \texttt{OK}(vs), \mu_3 \\ \mu_3 = \langle a, m_3, \sigma_3, l_3^f, l_3^m \rangle \\ \mathit{external\_call}(\sigma_3, a, v, fn) = v', \sigma_4 \qquad \mu_4 = \langle a, m_3, \sigma_4, l_3^f, l_3^m \rangle \end{array}}{p \vdash e_1(e_2{*}), \mu_1 \Rightarrow \texttt{OK}(v'), \mu_4}$$

An external call may arbitrarily change the state of the network, including the invoking account. This is a important source of security issues [6], as well as a problem for verification, since we lose any knowledge about the state we had up to this point. However this is not a new problem, as it has long been known in the context of verification of object-oriented programs [11, 3, 10, 13]. Several methodologies for reasoning about invariants across such calls have been proposed. Perhaps the simplest one is *visible-state semantics*, which would involve enforcing the invariants at all external call sites [11].

Modifiers and the placeholder statement. Next we give the semantics of the placeholder statement and function modifier execution. This rule may be triggered

[2] For simplicity function overloading is not taken into account here. To make it work, Solidity ABI uses hashes of the function signature, instead of just names and this is a mechanism that we do implement in Coq.

either as a part of the function call rules above, or when a placeholder statement is encountered inside a modifier. Once we are there, one of two possibilities may arise. The first one is that we have no more modifiers to execute, so we set the modifier local store to empty and enter the function body. This case is described by the following judgment:

$$\frac{\begin{array}{c}\mu = \langle a, m, \sigma, l^f, l^m \rangle \qquad \mu' = \langle a, m, \sigma, l^f, l_{empty} \rangle \\ p, \epsilon, f \vdash \mathit{body}(f), \mu' \Rightarrow o, \mu'' \qquad o \neq \mathtt{Fail}\end{array}}{p, \epsilon, f \vdash \_;, \mu \Rightarrow \mathtt{OK}(\epsilon), \mu''}$$

where the function *body* extracts the body of a given function. The second possibility is that we still have modifiers to execute:

$$\frac{\begin{array}{c}\mu_1 = \langle a, m_1, \sigma_1, l^f_1, l^m_1 \rangle \qquad \mu_2 = \langle a, m_1, \sigma_1, l^f_1, l_{empty} \rangle \\ p \vdash e*, \mu_2 [\Rightarrow] \mathtt{OK}(vs), \mu_3 \qquad \mu_3 = \langle a, m_3, \sigma_3, l^f_3, l_{empty} \rangle \\ \mu_4 = \langle a, m_3, \sigma_3, l^f_3, \mathit{mklocals}(\mathit{fm}, vs) \rangle \\ p, q', f \vdash \mathit{body}(\mathit{fm}), \mu_4 \Rightarrow o, \mu_5 \qquad o \neq \mathtt{Fail}\end{array}}{p, \langle \mathit{fm}, e* \rangle :: q, rv \vdash \_;, \mu_1 \Rightarrow \mathtt{OK}(\epsilon), \mu_5}$$

where $\langle \mathit{fm}, e* \rangle :: q$ is a notation for list pattern matching. We evaluate the arguments, create a new modifier local store and fill it with default values of the next modifier's local variables, pop the next modifier off the modifier stack and enter its body.

Having a separate store for function and modifier locals allows us to ensure that if the actual function body is entered more than once, the values of function's local variables are preserved, in accordance with the observed behavior produced by the Solidity compiler.

Return statement. The return statement is simple, but still interesting because of the return variable mechanism and its interaction with modifiers. The rule assigns values to the return variables and interrupts the control flow by using the `Return` outcome.

$$\frac{\begin{array}{c}p \vdash e*, \mu [\Rightarrow] \mathtt{OK}(vs), \mu' \qquad \mu' = \langle \ldots, l^f, \ldots \rangle \\ \mu'' = \langle \ldots, l^f[\mathit{retvar}_1(f) \to vs_1, \ldots, \mathit{retvar}(f) \to vs_n], \ldots \rangle\end{array}}{p, q, f \vdash \mathtt{return}\ e*;, \mu \Rightarrow \mathtt{Return}, \mu''}$$

Local variables. Local variable scoping follows Solidity versions prior to 0.5.0, i.e. there is a single scope for the entire function body. The lvalue rule for local variable is very simple:

$$\frac{}{p \vdash var, \mu \Leftarrow \mathtt{OK}(\mathtt{LVlocal}(var)), \mu}$$

There are two rules for dereferencing locals, one for modifiers, and one for functions:

$$\frac{p \vdash e, \mu \Leftarrow \mathtt{OK}(\mathtt{LVlocal}(var)), \mu \qquad s = \langle a, m, \sigma, l^f, l^m \rangle \qquad l^m(var) = \mathtt{Some}(v)}{p \vdash e, \mu \Rightarrow \mathtt{OK}(v), \mu}$$

$$\frac{\begin{array}{cc} p \vdash e, \mu \Leftarrow \mathtt{OK}(\mathtt{LVlocal}(\mathit{var})), \mu & s = \langle a, m, \sigma, l^f, l^m \rangle \\ l^m(\mathit{var}) = \mathtt{None} & l^f(\mathit{var}) = \mathtt{Some}(v) \end{array}}{p \vdash e, \mu \Rightarrow \mathtt{OK}(v), \mu}$$

Local variables are first looked up in the modifier store and then, if this fails, in the function store. This could cause functions locals to be shadowed by modifier locals, but we ensure that this is not the case by setting the modifier store to empty when entering a function body. This trick allows us to forgo adding information about whether our current expression is evaluated within a function or modifier body. The possibility of accessing function locals from modifiers should be ruled out during the typechecking phase.

State variables and storage. Now we show the rules dealing with state variables and storage objects. The lvalue rules are straightforward:

$$\frac{}{p \vdash \mathit{var}\{\mathit{cname}\}, \mu \Leftarrow \mathtt{OK}(\mathtt{LVstorage}(\mathtt{SRvar}(\mathit{var}, \mathit{cname}))), \mu}$$

$$\frac{p \vdash e, \mu \Rightarrow \mathtt{OK}(\mathtt{Vsptr}(\mathit{sr})), \mu'}{p \vdash e.\mathit{fname}, \mu \Leftarrow \mathtt{OK}(\mathtt{LVstorage}(\mathtt{SRstruct\_field}(\mathit{sr}, \mathit{fname}))), \mu'}$$

Analogous rules exist for element access of arrays or mappings.

The rules for dereferencing storage objects are as follows:

$$\frac{p \vdash e, \mu \Leftarrow \mathtt{OK}(\mathtt{LVstorage}(\mathit{sr})), \mu \quad \mu = \langle \ldots, s, \ldots \rangle \quad s(\mathit{sr}) = \mathtt{Some}(\mathtt{Sval}(v))}{p \vdash e, \mu \Rightarrow \mathtt{OK}(v), \mu}$$

$$\frac{p \vdash e, \mu \Leftarrow \mathtt{OK}(\mathtt{LVstorage}(\mathit{sr})), \mu \quad \mu = \langle \ldots, s, \ldots \rangle \quad s(\mathit{sr}) \neq \mathtt{Some}(\mathtt{Sval}(\_))}{p \vdash e, \mu \Rightarrow \mathtt{OK}(\mathtt{Vsref}(\mathit{sr})), \mu}$$

When the storage reference points to an `Sval`, the value it contains is returned. Otherwise a pointer is returned.

Assigning to a storage location in Solidity is quite complicated. Consider the rule for assigning from another storage location:

$$\frac{\begin{array}{ccc} p \vdash e_1, \mu_1 \Leftarrow \mathtt{OK}(\mathtt{LVstorage}(\mathit{sr}_1)), \mu_2 & & p \vdash e_2, \mu_2 \Rightarrow \mathtt{OK}(\mathtt{Vsref}\,\mathit{sr}_2), \mu_3 \\ \mu_3 = \langle a, m_3, \sigma_3, l^f_3, l^m_3 \rangle & & \sigma_3(a) = \langle b, p, s \rangle \\ s(\mathit{sr}_1) = \mathtt{Some}(\mathit{so}_1) & s(\mathit{sr}_2) = \mathtt{Some}(\mathit{so}_2) & \mathit{copy\_over}(\mathit{so}_1, \mathit{so}_2) = \mathtt{Some}(\mathit{so}_3) \\ s' = s[\mathit{sr}_1 \to \mathit{so}_3] & & \mu_4 = \langle a, m_3, \sigma_3[a \to \langle b, p, s', l^f_3, l^m_3 \rangle] \rangle \end{array}}{p \vdash e_1 = e_2, \mu \Rightarrow \mathtt{OK}(\mathit{storage\_dereference}(\mathit{sr}_1, s')), \mu_4}$$

As mentioned before, assigning a storage object to a storage location causes deep copying into that location. Intuitively, it should be sufficient to replace the storage object so_1 pointed to by sr_1 with so_2. Sadly, the inability to copy contents of mappings introduces an ugly corner case: when copying so_2 object over so_1, the contents of mappings reachable from so_1 are preserved. That behavior is modeled by the function *copy_over*.

3.2 Current state of the Coq development

As mentioned, we have written down our semantics in an executable form in Coq. This way, it can be combined with code that parses and typechecks Solidity to enable execution of basic contract code. Currently, a very rudimentary test environment has been implemented that enables running simple contract functions, such as the examples given in Section 2, except those involving external calls.

Solidity is quite a large language and a significant amount of work is still to be done claim any completeness or run real-world contracts. In particular, our typechecker is written in a mostly ad-hoc manner in Ocaml, still largely incomplete and limited to features necessary for execution, such as resolving state variable names and expression types. Other features of the type system, such as visibility and mutability specifiers are ignored. Solidity provides a number of syntactic sugar constructs, such as named arguments, that we left out. We still do not support the full range of available types and builtin functions. For example, support for small integer types, strings, and packed byte arrays is incomplete. Libraries (accounts containing reusable code), an another widely used feature yet to be formalized, are an significant omission, especially since their semantics can also be a source of serious problems [16]. Some features, like inline assembly, are incompatible with our high-level modeling and cannot be implemented.

Once we implement enough of the language, to establish trust in the specification, we plan to test the semantics against the Solidity compiler test suite. The way we plan to do this, is by implementing a mock Ethereum client with an RPC API that accepts Solidity code instead of EVM bytecode and modifying the Solidity test suite to work with it. The work on this infrastructure has been started, but it is not yet fully functional.

4 Related work

Bhargavan et al. [5] provided a verification framework for a subset of Solidity by the way of shallow embedding in F*, a programming language aimed at program verification. However they have not provided explicit semantics, and we could not reproduce the data model of Solidity from the descriptions thereof.

Much work has been put into formally specifying the semantics of the Ethereum Virtual Machine. Hirai [9] defined EVM semantics in Lem, a language that can be compiled into specifications for several theorem provers, and then used it to prove safety properties of smart contracts. His formalization was extended with a program logic by Amani et al. [1]. Hildenbrandt et al. [8] defined complete executable EVM semantics in the K Framework, which passed the reference test suite for EVM implementations. Luu et al. have created Oyente [12], a static analysis tool for EVM bytecode. For that purpose, they have developed a simplified semantics of a fragment of EVM. Grishchenko et al. [7] present complete small-step semantics of EVM bytecode, formalized in F*.

Sergey et al. [18] describe Scilla, an intermediate-level programming language for smart contracts that aims to provide clear operational semantics. They restrict the computation model to communicating automata and mandate external calls to occur at the end of a transaction. This makes the language more amenable to formal verification techniques.

5 Conclusions and future work

We have presented a formalization of what we consider to be the core of Solidity, in the form of big-step semantics. We have focused on high-level modeling of the data model and semantics of internal function calls with function modifiers. We have written down our semantics in an executable form in Coq.

Many features used in real contracts still remain to be formalized. In the near future we plan to specify semantics of a larger subset of Solidity. To establish trust in the specification, the executable semantics could then be tested against the official Solidity compiler test suite. Additionally, a coming release of Solidity (0.5.0) is planned to bring many changes to the language, like C99-like block scoping for local variables, and our semantics has to be adapted accordingly.

Verification of realistic contracts is still a somewhat distant goal. We do not consider raw operational semantics to be a practical tool for verification of contracts, for example due to axiomatization of external calls. Ultimately, our aim with this work is to provide a foundation for verification frameworks for smart contracts written in Solidity.

Acknowledgements I would like to thank Aleksy Schubert for his helpful comments on the draft versions of this paper.

References

1. Amani, S., Bégel, M., Bortin, M., Staples, M.: Towards verifying Ethereum smart contract bytecode in Isabelle/HOL. In: Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs. pp. 66–77. CPP 2018, ACM, New York, NY, USA (2018). https://doi.org/10.1145/3167084, http://doi.acm.org/10.1145/3167084
2. 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 New York, Inc., New York, NY, USA (2017). https://doi.org/10.1007/978-3-662-54455-6_8, https://doi.org/10.1007/978-3-662-54455-6_8
3. Barnett, M., DeLine, R., Fahndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. JOURNAL OF OBJECT TECHNOLOGY **3**, 2004 (2003)
4. Barrett, K., Cassels, B., Haahr, P., Moon, D.A., Playford, K., Withington, P.T.: A monotonic superclass linearization for Dylan. In: Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications. pp. 69–82. OOPSLA '96, ACM, New York, NY, USA (1996). https://doi.org/10.1145/236337.236343, http://doi.acm.org/10.1145/236337.236343
5. 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: ACM Workshop on Programming Languages and Analysis for Security. Vienna, Austria (Oct 2016). https://doi.org/10.1145/2993600.2993611, https://hal.inria.fr/hal-01400469
6. Buterin, V.: CRITICAL UPDATE Re: DAO Vulnerability. Retrieved April 24, 2018, from https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability/

7. 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 International Publishing, Cham (2018)
8. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Rosu, G.: KEVM: A complete semantics of the Ethereum Virtual Machine. Tech. rep. (2017)
9. 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 International Publishing, Cham (2017)
10. Leino, K.R.M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004 – Object-Oriented Programming. pp. 491–515. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)
11. Leino, K.R.M., Stata, R.: Checking object invariants (1997)
12. Luu, L., Chu, D.H., Olickel, H., Saxena, P., Hobor, A.: Making smart contracts smarter. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 254–269. CCS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2976749.2978309, http://doi.acm.org/10.1145/2976749.2978309
13. Naumann, D.A., Barnett, M.: Towards imperative modules: Reasoning about invariants and sharing of mutable state. Theoretical Computer Science **365**(1), 143 – 168 (2006). https://doi.org/https://doi.org/10.1016/j.tcs.2006.07.035, http://www.sciencedirect.com/science/article/pii/S0304397506004841, formal Methods for Components and Objects
14. Nipkow, T.: Jinja: Towards a comprehensive formal semantics for a Java-like language. In: Schwichtenberg, H., Spies, K. (eds.) Proof Technology and Computation. pp. 247–277. IOS Press (2006)
15. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632. pp. 589–615. Springer-Verlag New York, Inc., New York, NY, USA (2016). https://doi.org/10.1007/978-3-662-49498-1_23, http://dx.doi.org/10.1007/978-3-662-49498-1_23
16. A Postmortem on the Parity Multi-Sig Library Self-Destruct. Retrieved April 24, 2018, from https://paritytech.io/a-postmortem-on-the-parity-multi-sig-library-self-destruct/
17. Ramananandro, T., Dos Reis, G., Leroy, X.: Formal verification of object layout for C++ multiple inheritance. In: Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 67–80. POPL '11, ACM, New York, NY, USA (2011). https://doi.org/10.1145/1926385.1926395, http://doi.acm.org/10.1145/1926385.1926395
18. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. CoRR **abs/1801.00687** (2018)
19. Solidity documentation. Retrieved April 22, 2018, from https://solidity.readthedocs.io/en/develop/index.html