SMT-Friendly Formalization of the Solidity Memory Model

Ákos Hajdu[1]* and Dejan Jovanović[2]

[1] Budapest University of Technology and Economics, Budapest, Hungary
hajdua@mit.bme.hu
[2] SRI International, New York City, USA
dejan.jovanovic@sri.com

Abstract. Solidity is the dominant programming language for Ethereum smart contracts. This paper presents a high-level formalization of the Solidity language with a focus on the memory model. The presented formalization covers all features of the language related to managing state and memory. In addition, the formalization we provide is effective: all but few features can be encoded in the quantifier-free fragment of standard SMT theories. This enables precise and efficient reasoning about the state of smart contracts written in Solidity. The formalization is implemented in the SOLC-VERIFY verifier and we provide an extensive set of tests that covers the breadth of the required semantics. We also provide an evaluation on the test set that validates the semantics and shows the novelty of the approach compared to other Solidity-level contract analysis tools.

1 Introduction

Ethereum [32] is a public blockchain platform that provides a novel computing paradigm for developing decentralized applications. Ethereum allows the deployment of arbitrary programs (termed smart contracts [31]) that operate over the blockchain state. The public can interact with the contracts via transactions. It is currently the most popular public blockchain with smart contract functionality. While the nodes participating in the Ethereum network operate a low-level, stack-based virtual machine (EVM) that executes the compiled smart contracts, the contracts themselves are mostly written in a high-level, contract-oriented programming language called Solidity [30].

Even though smart contracts are generally short, they are no less prone to errors than software in general. In the Ethereum context, any flaws in the contract code come with potentially devastating financial consequences (such as the infamous DAO exploit [17]). This has inspired a great interest in applying formal verification techniques to Ethereum smart contracts (see e.g., [4] or [14] for surveys). In order to apply formal verification of any kind, be it static analysis or

---

model checking, the first step is to formalize the semantics of the programming language that the smart contracts are written in. Such semantics should not only remain an exercise in formalization, but should preferably be developed, resulting in precise and automated verification tools.

Early approaches to verification of Ethereum smart contracts focused mostly on formalizing the low-level virtual machine precisely (see, e.g., [11,19,21,22,2]). However, the unnecessary details of the EVM execution model make it difficult to reason about high-level functional properties of contracts (as they were written by developers) in an effective and automated way. For Solidity-level properties of smart contracts, Solidity-level semantics are preferred. While some aspects of Solidity have been studied and formalized [23,10,15,33], the semantics of the Solidity memory model still lacks a detailed and precise formalization that also enables automation.

The memory model of Solidity has various unusual and non-trivial behaviors, providing a fertile ground for potential bugs. Smart contracts have access to two classes of data storage: a permanent storage that is a part of the global blockchain state, and a transient local memory used when executing transactions. While the local memory uses a standard heap of entities with references, the permanent storage has pure value semantics (although pointers to storage can be declared locally). This memory model that combines both value and reference semantics, with all interactions between the two, poses some interesting challenges but also offers great opportunities for automation. For example, the value semantics of storage ensures non-aliasing of storage data. This can, if supported by an appropriate encoding of the semantics, potentially improve both the precision and effectiveness of reasoning about contract storage.

This paper provides a formalization of the Solidity semantics in terms of a simple SMT-based intermediate language that covers all features related to managing contract storage and memory. A major contribution of our formalization is that all but few of its elements can be encoded in the quantifier-free fragment of standard SMT theories. Additionally, our formalization captures the value semantics of storage with implicit non-aliasing information of storage entities. This allows precise and effective verification of Solidity smart contracts using modern SMT solvers. The formalization is implemented in the open-source SOLC-VERIFY tool [20], which is a modular verifier for Solidity based on SMT solvers. We validate the formalization and demonstrate its effectiveness by evaluating it on a comprehensive set of tests that exercise the memory model. We show that our formalization significantly improves the precision and soundness compared to existing Solidity-level verifiers, while remarkably outperforming low-level EVM-based tools in terms of efficiency.

2   Background

2.1   Ethereum

Ethereum [32,3] is a generic blockchain-based distributed computing platform. The Ethereum ledger is a storage layer for a database of accounts (identified

by addresses) and the data associated with the accounts. Every account has an associated balance in Ether (the native cryptocurrency of Ethereum). In addition, an account can also be associated with the executable bytecode of a contract and the contract state.

Although Ethereum contracts are deployed to the blockchain in the form of the bytecode of the Ethereum Virtual Machine (EVM) [32], they are generally written in a high-level programming language called Solidity [30] and then compiled to EVM bytecode. After deployment, the contract is publicly accessible and its code cannot be modified. An external user, or another contract, can interact with a contract through its API by invoking its public functions. This can be done by issuing a transaction that encodes the function to be called with its arguments, and contains the contract's address as the recipient. The Ethereum network then executes the transaction by running the contract code in the context of the contract instance.

A contract instance has access to two different kinds of memory during its lifetime: contract storage and memory.[3] *Contract storage* is a dedicated data store for a contract to store its persistent state. At the level of the EVM, it is an array of 256-bit storage *slots* stored on the blockchain. Contract data that fits into a slot, or can be sliced into fixed number of slots, is usually allocated starting from slot 0. More complex data types that do not fit into a fixed number of slots, such as mappings, or dynamic arrays, are not supported directly by the EVM. Instead, they are implemented by the Solidity compiler using storage as a hash table where the structured data is distributed in a deterministic collision-free manner. *Contract memory* is used during the execution of a transaction on the contract, and is deleted after the transaction finishes. This is where function parameters, return values and temporary data can be allocated and stored.

2.2   Solidity

Solidity [30] is the high-level programming language supporting the development of Ethereum smart contracts. It is a full-fledged object-oriented programming language with many features focusing on enabling rapid development of Ethereum smart contracts. The focus of this paper is the semantics of the Solidity memory model: the Solidity view of contract storage and memory, and the operations that can modify it. Thus, we restrict the presentation to a generous fragment of Solidity that is relevant for discussing and formalizing the memory model. An example contract that illustrates relevant features is shown in Figure 1, and the abstract syntax of the targeted fragment is presented in Figure 2. We omit parts of Solidity that are not relevant to the memory model (e.g., inheritance, loops, blockchain-specific members). We also omit low-level, unsafe features that can break the Solidity memory model abstractions (e.g., `assembly` and `delegatecall`).

---

[3] There is an additional data location named *calldata* that behaves the same as memory, but is used to store parameters of external functions. For simplicity, we omit it in this paper.

```
contract DataStorage {
  struct Record {
    bool set;
    int[] data;
  }

  mapping(address=>Record) private records;

  function append(address at, int d) public {
    Record storage r = records[at];
    r.set = true;
    r.data.push(d);
  }
  function isset(Record storage r) internal view returns (bool s) {
    s = r.set;
  }
  function get(address at) public view returns (int[] memory ret) {
    require(isset(records[at]));
    ret = records[at].data;
  }
}
```

Fig. 1: An example contract illustrating commonly used features of the Solidity memory model. The contract keeps an association between addresses and data and allows users to query and append to their data.

Contracts. Solidity contracts are similar to classes in object-oriented programming. A contract can define any additional types needed, followed by the declaration of the *state variables* and contract *functions*, including an optional single *constructor* function. The contract's state variables define the only persistent data that the contract instance stores on the blockchain. The constructor function is only used once, when a new contract instance is deployed to the blockchain. Other public contract functions can be invoked arbitrarily by external users through an Ethereum transaction that encodes the function call data and designates the contract instance as the recipient of the transaction.

Example 1. The contract `DataStorage` in Figure 1 defines a struct type `Record`. Then it defines the contract storage as a single state variable `records`. Finally three contract functions are defined `append()`, `isset()`, and `get()`. Note that a constructor is not defined and, in this case, a default constructor is provided to initialize the contract state to default values.

Solidity supports further concepts from object-oriented programming, such as inheritance, function modifiers, and overloading (also covered by our implementation [20]). However, as these are not relevant for the formalization of the memory model we omit them to simplify our presentation.

Types. Solidity is statically typed and provides two classes of types: *value* types and *reference* types. Value types include elementary types such as addresses, integers, and Booleans that are always passed by value. Reference types, on the other hand, are passed by reference and include structs, arrays and mappings.

TypeName	::= `address` | `int` | `uint` | `bool`	Value types
	| `mapping(` *TypeName* `=>` *TypeName* `)`	Mapping
	| *TypeName* `[]` | *TypeName* `[n]`	Arrays
	| *StructName*	Struct name
DataLoc	::= `storage` | `memory`	Data location
lval	::= *id*	Identifier
	| *expr* . *id*	Member access
	| *expr* [*expr*]	Index access
expr	::= *lval*	Lvalue
	| *expr* ? *expr* : *expr*	Conditional
	| `new` *TypeName* `[](`*expr*`)`	New memory array
	| *StructName* (*expr**)	New memory struct
stmt	::= *TypeName DataLoc*? *id* [= *expr*];	Local variable declaration
	| (*lval*)* = (*expr*)*;	Assignment (tuples)
	| *lval*.`push(`*expr*`)`;	Push
	| *lval*.`pop()`;	Pop
	| `delete` *lval*;	Delete
StructMem	::= *TypeName id*;	Struct member
StructDef	::= `struct` *StructName* { *StructMem** }	Struct definition
StateVar	::= *TypeName id*;	State variable definition
FunPar	::= *TypeName DataLoc*? *id*	Function parameter
Fun	::= `function` *id*(*FunPar**)	Function definition
	[`returns` (*FunPar**)] { *stmt** }	
Constr	::= `constructor`(*FunPar**) { *stmt** }	Constructor definition
Contract	::= `contract` *id*	Contract definition
	{*StructDef** *StateVar** *Constr*? *Fun**}	

Fig. 2: Syntax of the targeted Solidity fragment.

A struct consists of a fixed number of members. An array is either fixed-size or dynamically-sized and besides the elements of the base type, it also includes a `length` field holding the number of elements. A mapping is an associative array mapping keys to values. The important caveat is that the table does not actually store the keys so it is not possible to check if a key is defined in the map.

Example 2. The contract in Figure 1 uses the following types. The `records` variable is a mapping from addresses to `Record` structures which, in turn, consist of a Boolean value and a dynamically-sized integer array. It is a common practice to define a struct with a Boolean member (`set`) to indicate that a mapping value has been set. This is because Solidity mappings do not store keys: any key can be queried, returning a default value if no value was associated previously.

Data locations for reference types. Data of reference types resides in a *data location* that is either *storage* or *memory*. Storage is the persistent store used for state variables of the contract. In contrast, memory is used during execution of a transaction to store function parameters, return values and local variables, and it is deleted after the transaction finishes.

Semantics of reference types differ fundamentally depending on the data location that they are stored in. Layout of data in the memory data location resembles the memory model common in Java-like programming languages: there is a heap where reference types are allocated and any entity in the heap can contain values of value types, and *references* to other memory entities. In contrast, the storage data location treats and stores all entities, including those of reference types, as *values* with no references involved. Mixing storage and memory is not possible: the data location of a reference type is propagated to its elements and members. This means that storage entities cannot have references to memory entities, and memory entities cannot have reference types as values. Storage of a contract can be viewed as a single value with no aliasing possible.
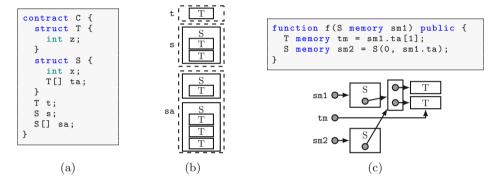

(a)   (b)   (c)

Fig. 3: An example illustrating reference types (structs and arrays) and their layout in storage and memory: (a) a contract defining types and state variables; (b) an abstract representation of the contract storage as values; and, (c) a function using the memory data location and a possible layout of the data in memory.

Example 3. Consider the contract C defined in Figure 3a. The contract defines two reference struct types S and T, and declares state variables s, t, and sa. These variables are maintained in storage during the contract lifetime and they are represented as values with no references within. A potential value of these variables is shown in Figure 3b. On the other hand, the top of Figure 3c shows a function with three variables in the memory data location, one as the argument to the function, and two defined within the function. Because they are in memory, these variables are references to heap locations. Any data of reference types, stored within the structures and arrays, is also a reference and can be reallocated or assigned to point to an existing heap location. This means that the layout of the data can contain arbitrary graphs with arbitrary aliasing. A potential layout of these variables is shown at the bottom of Figure 3c.

Functions. Functions are the Solidity equivalent of methods in classes. They receive data as arguments, perform computations, manipulate state variables

and interact with other Ethereum accounts. Besides accessing the storage of the contract through its state variables, functions can also define local variables, including function arguments and return values. Variables of value types are stored as values on a stack. Variables of reference types must be explicitly declared with a data location, and are always pointers to an entity in that data location (storage or memory). A pointer to storage is called a *local storage pointer*. As the storage is not memory in the usual sense, but a value instead, one can see storage pointers as encoding a path to one reference type entity in the storage.

Example 4. Consider the example in Figure 1. The local variable `r` in function `append()` points to the struct at index `at` of the state variable `records` (residing in the contract storage). In contrast, the return value `ret` of function `get()` is a pointer to an integer array in memory.

Statements and expressions. Solidity includes usual programming statements and control structures. To keep the presentation simple, we focus on the statements that are related to the formalization of the memory model: local variable declarations, assignments, array manipulation, and the `delete` statement.[4] Solidity expressions relevant for the memory model are identifiers, member and array accesses, conditionals and allocation of new arrays and structs in memory.

If a value is not provided, local variable declarations automatically initialize the variable to a default value. For reference types in memory, this allocates new entities on the heap and performs recursive initialization of its members. For reference types in storage, the local storage pointers must always be explicitly initialized to point to a storage member. This ensures that no pointer is ever "null". Value types are initialized to their simple default value (0, false). Behavior of assignment in Solidity is complex (see Section 3.5) and depends on the data location of its arguments (e.g., deep copy or pointer assignment). Dynamically-sized storage arrays can be extended by pushing an element to their end, or can be shrunk by popping. The `delete` statement assigns the default value (recursively for reference types) to a given entity based on its type.

Example 5. The assignment `r.set = true` in the `append()` function of Figure 1 is a simple value assignment. On the other hand, `ret = records[at].data` in the `get()` function allocates a new array on the heap and performs a deep copy of data from storage to memory.

2.3   SMT-Based Programs

We formalize the semantics of the Solidity fragment by translating it to a simple programming language that uses SMT semantics [9,12] for the types and data. The syntax of this language is shown in Figure 4. The syntax is purposefully

---

[4] Our implementation [20] supports a majority of statements, excluding low-level operations (such as inline assembly). Loops are also supported and can be specified with loop invariants.

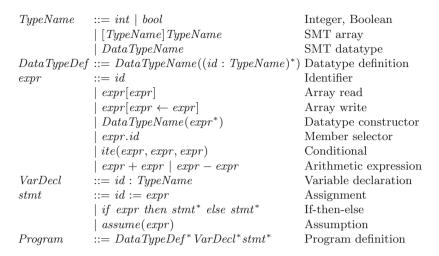| $TypeName$	$::= int \mid bool$	Integer, Boolean
	$\mid [TypeName]\, TypeName$	SMT array
	$\mid DataTypeName$	SMT datatype
$DataTypeDef$	$::= DataTypeName((id : TypeName)^*)$	Datatype definition
$expr$	$::= id$	Identifier
	$\mid expr[expr]$	Array read
	$\mid expr[expr \leftarrow expr]$	Array write
	$\mid DataTypeName(expr^*)$	Datatype constructor
	$\mid expr.id$	Member selector
	$\mid ite(expr, expr, expr)$	Conditional
	$\mid expr + expr \mid expr - expr$	Arithmetic expression
$VarDecl$	$::= id : TypeName$	Variable declaration
$stmt$	$::= id := expr$	Assignment
	$\mid if\ expr\ then\ stmt^*\ else\ stmt^*$	If-then-else
	$\mid assume(expr)$	Assumption
$Program$	$::= DataTypeDef^*\, VarDecl^*\, stmt^*$	Program definition

Fig. 4: Syntax of SMT-based programs.

minimal and generic, so that it can be expressed in any modern SMT-based verification tool (e.g., Boogie [5], Why3 [18] or Dafny [26]).[5]

The types of SMT-based programs are the SMT types: simple value types such as Booleans and mathematical integers, and structured types such as arrays [27,16] and inductive datatypes [8]. The expressions of the language are standard SMT expressions such as identifiers, array reads and writes, datatype constructors, member selectors, conditionals and basic arithmetic [7]. All variables are declared at the beginning of a program. The statements of the language are limited to assignments, the if-then-else statement, and assumption statement.

SMT-based programs are a good fit for modeling of program semantics. For one, they have clear semantics with no ambiguities. Furthermore, any property of the program can be checked with SMT solvers: the program can be translated directly to a SMT formula by a single static assignment (SSA) transformation.

Note that the syntax requires the left hand side of an assignment to be an identifier. However, to make our presentation simpler, we will allow array read, member access and conditional expressions (and their combination) as LHS. Such constructs can be eliminated iteratively in the following way until only identifiers appear as LHS in assignments.

- $a[i] := e$ is equivalent to $a := a[i \leftarrow e]$.
- $d.m_j := e$ is equivalent to $d := D(d.m_1, \ldots, d.m_{j-1}, e, d.m_{j+1}, \ldots, d.m_n)$, where D is the constructor of a datatype with members $m_1, \ldots, m_n$.
- $ite(c, t, f) := e$ is equivalent to $if\ c\ then\ t := e\ else\ f := e$.

---

[5] Our current implementation is based on Boogie, but we have plans to introduce a generic intermediate representation that could incorporate alternate backends such as Why3 or Dafny.

3   Formalization

In this section we present our formalization of the Solidity semantics through a translation that maps Solidity elements to constructs in the SMT-based language. The formalization is described top-down in separate subsections for types, contracts, state variables, functions, statements, and expressions.

3.1   Types

We use $\mathcal{T}(.)$ to denote the function that maps a Solidity type to an SMT type. This function is used in the translation of contract elements and can, as a side effect, introduce datatype definitions and variable declarations. This is denoted with $[decl]$ in the result of the function. To simplify the presentation, we assume that such side effects are automatically added to the preamble of the SMT program. Furthermore, we assume that declarations with the same name are only added once. We use $\mathsf{type}(expr)$ to denote the original (Solidity) type of an expression (to be used later in the formalization). The definition of $\mathcal{T}(.)$ is shown in Figure 5.

$$\mathcal{T}(\texttt{bool}) \quad \doteq bool$$
$$\mathcal{T}(\texttt{address}) \doteq \mathcal{T}(\texttt{int}) \doteq \mathcal{T}(\texttt{uint}) \doteq int$$

$$\mathcal{T}(\texttt{mapping}(K\texttt{=>}V) \texttt{ storage}) \doteq [\mathcal{T}(K)]\mathcal{T}(V)$$
$$\mathcal{T}(\texttt{mapping}(K\texttt{=>}V) \texttt{ storptr}) \doteq [int]int$$

$$\mathcal{T}(T[n] \texttt{ storage}) \doteq \mathcal{T}(T[] \texttt{ storage})$$
$$\mathcal{T}(T[n] \texttt{ storptr}) \doteq \mathcal{T}(T[] \texttt{ storptr})$$
$$\mathcal{T}(T[n] \texttt{ memory}) \doteq \mathcal{T}(T[] \texttt{ memory})$$

$$\mathcal{T}(T[] \texttt{ storage}) \doteq StorArr_T \text{ with } [StorArr_T(arr : [int]\mathcal{T}(T), length : int)]$$
$$\mathcal{T}(T[] \texttt{ storptr}) \doteq [int]int$$
$$\mathcal{T}(T[] \texttt{ memory}) \doteq int \qquad \text{with } [MemArr_T(arr : [int]\mathcal{T}(T), length : int)]$$
$$\qquad\qquad\qquad\qquad\qquad [arrheap_T : [int]MemArr_T]$$

$$\mathcal{T}(\texttt{struct } S \texttt{ storage}) \doteq StorStruct_S \text{ with } [StorStruct_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]$$
$$\mathcal{T}(\texttt{struct } S \texttt{ storptr}) \doteq [int]int$$
$$\mathcal{T}(\texttt{struct } S \texttt{ memory}) \doteq int \qquad \text{with } [MemStruct_S(\ldots, m_i : \mathcal{T}(S_i), \ldots)]$$
$$\qquad\qquad\qquad\qquad\qquad [structheap_S : [int]MemStruct_S]$$

Fig. 5: Formalization of Solidity types. Members of struct S are denoted as m_i with types S_i.

Value types. Booleans are mapped to SMT Booleans while other value types are mapped to SMT integers. Addresses are also mapped to SMT integers so that arithmetic comparison and conversions between integers and addresses is supported. For simplicity, we map all integers (signed or unsigned) to SMT

integers.[6] Solidity also allows function types to store, pass around, and call functions, but this is not yet supported by our encoding.

Reference types. The Solidity syntax does not always require the data location for variable and parameter declarations. However, for reference types it is always required (enforced by the compiler), except for state variables that are always implicitly storage. In our formalization, we assume that the data location of reference types is a part of the type. As discussed before, memory entities are always accessed through pointers. However, for storage we distinguish whether it is the storage reference itself (e.g., state variable) or a storage pointer (e.g., local variable, function parameter). We denote the former with `storage` and the latter with `storptr` in the type name. Our modeling of reference types relies on the generalized theory of arrays [16] and the theory of inductive data-types [8], both of which are supported by modern SMT solvers (e.g., CVC4 [6] and z3 [28]).

Mappings and arrays. For both arrays and mappings, we abstract away the implementation details of Solidity and model them with the SMT theory of arrays and inductive datatypes. We formalize Solidity mappings simply as SMT arrays. Both fixed- and dynamically-sized arrays are translated using the same SMT type and we only treat them differently in the context of statements and expressions. Strings and byte arrays are not discussed here, but we support them as particular instances of the array type. To ensure that array size is properly modeled we keep track of it in the datatype ($length$) along with the actual elements (arr).

For *storage array types* with base type T, we introduce an SMT datatype $StorArr_T$ with a constructor that takes two arguments: an inner SMT array (arr) associating integer indexes and the recursively translated base type ($\mathcal{T}(T)$), and an integer *length*. The advantage of this encoding is that the value semantics of storage data is provided by construction: each array element is a separate entity (no aliasing) and assigning storage arrays in SMT makes a deep copy. This encoding also generalizes if the base type is a reference type.

For *memory array types* with base type T, we introduce a separate datatype $MemArr_T$ (side effect). However, memory arrays are stored with pointer values. Therefore the memory array type is mapped to integers, and a heap ($arrheap_T$) is introduced to associate integers (pointers) with the actual memory array datatypes. Note that mixing data locations within a reference type is not possible: the element type of the array has the same data location as the array itself. Therefore, it is enough to introduce two datatypes per element type T: one for storage and one for memory. In the former case the element type will have value semantics whereas in the latter case elements will be stored as pointers.

Structs. For each *storage struct type* S the translation introduces an inductive datatype $StorStruct_S$, including a constructor for each struct member with types

---

[6] Note that this does not capture the precise machine integer semantics, but this is not relevant from the perspective of the memory model. Precise computation can be provided by relying on SMT bitvectors or modular arithmetic (see, e.g., [20]).

mapped recursively. Similarly to arrays, this ensures the value semantics of storage such as non-aliasing and deep copy assignments. For each *memory struct S* we also introduce a datatype $MemStruct_S$ and a constructor for each member.[7] However, the memory struct type itself is mapped to integers (pointer) and a heap ($structheap_S$) is introduced to associate the pointers with the actual memory struct datatypes. Note that if a memory struct has members with reference types, they are also pointers, which is ensured recursively by our encoding.

3.2   Local Storage Pointers

An interesting aspect of the storage data location is that, although the stored data has value semantics, it is still possible to define pointers to an entity in storage within a local context, e.g., with function parameters or local variables. These pointers are called *local storage pointers*.

Example 6. In the `append()` function of Figure 1 the variable `r` is defined to be a convenience pointer into the storage map `records[at]`. Similarly, the `isset()` function takes a storage pointer to a `Record` entity in storage as an argument.

Since our formalization uses SMT datatypes to encode the contract data in storage, it is not possible to encode these pointers directly. A partial solution would be to substitute each occurrence of the local pointer with the expression that is assigned to it when it was defined. However, this approach is too simplistic and has limitations. Local storage pointers can be reassigned, or assigned conditionally, or it might not be known at compile time which definition should be used. Furthermore, local storage pointers can also be passed in as function arguments: they can point to different storage entities for different calls.

   We propose an approach to encode local storage pointers while overcoming these limitations. Our encoding relies on the fact that storage data of a contract can be viewed as a finite-depth tree of values. As such, each element of the stored data can be uniquely identified by a finite path leading to it.[8]

Example 7. Consider the contract `C` in Figure 6a. The contract defines structs `T` and `S`, and state variables of these types. If we are interested in all storage entities of type `T`, we can consider the sub-tree of the contract storage tree that has leaves of type `T`, as depicted in Figure 6b. The root of the tree is the contract itself, with indexed sub-nodes for state variables, in order. For nodes of struct type there are indexed sub-nodes leading to its members, in order. For each node of array type there is a sub-node for the base type. Every pointer to a storage `T` entity can be identified by a path in this tree: by fixing the index to each state

---

[7] Mappings in Solidity cannot reside in memory. If a struct defines a mapping member and it is stored in memory, the mapping is simply inaccessible. Such members could be omitted from the constructor.

[8] Solidity does support a limited form of recursive data-types. Such types could make the storage a tree of potentially arbitrary depth. We chose not to support such types as recursion is non-existing in Solidity types used in practice.

```
contract C {
  struct T {
    int z;
  }
  struct S {
    int x;
    T    t;
    T[] ts;
  }
  T    t1;
  S    s1;
  S [] ss;
}
```

$$\begin{aligned}
\mathsf{unpack}(ptr) = \\
ite(ptr[0] = 0, \\
t1, \\
ite(ptr[0] = 1, \\
ite(ptr[1] = 0, \\
s1.t, \\
s1.ts[ptr[2]]), \\
ite(ptr[2] = 0, \\
ss[ptr[1]].t, \\
ss[ptr[1]].ts[ptr[3]])))
\end{aligned}$$

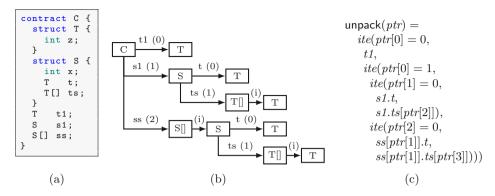(a)   (b)   (c)

Fig. 6: An example of packing and unpacking: (a) contract with struct definitions and state variables; (b) the storage tree of the contract for type T; and (c) the unpacking expression for storage pointers of type T.

variable, member, and array index, as seen in brackets in Figure 6b, such paths can be encoded as an array of integers. For example, the state variable `t1` can be represented as $[0]$, the member `s1.t` as $[1, 0]$, and `ss[8].ts[5]` as $[2, 8, 1, 5]$.

This idea allows us to encode storage pointer types (pointing to arrays, structs or mappings) simply as SMT arrays ($[int]int$). The novelty of our approach is that storage pointers can be encoded and passed around, while maintaining the value semantics of storage data, without the need for quantifiers to describe non-aliasing. To encode storage pointers, we need to address initialization and dereference of storage pointers, while assignment is simply an assignment of array values. When a storage pointer is initialized to a concrete expression, we *pack* the indexed path to the storage entity (that the expression references) into an array value. When a storage pointer is dereferenced (e.g., by indexing into or accessing a member), the array is *unpacked* into a conditional expression that will evaluate to a storage entity by decoding paths in the tree.

Storage tree. The storage tree for a given type T can be easily obtained by filtering the AST nodes of the contract definition to only include state variable declarations and to, further, only include nodes that lead to a sub-node of type T. We denote the storage tree for type T as $\mathsf{tree}(T)$.[9]

Packing. Given an expression (such as `ss[8].ts[5]`), pack(.) uses the storage tree for the type of the expression and encodes it to an array (e.g., $[2, 8, 1, 5]$) by fitting the expression into the tree. Pseudocode for pack(.) is shown in Figure 7. To start, the expression is decomposed into a list of base sub-expressions. The base expression of an identifier *id* is *id* itself. For an array index $e[i]$ or a member

---

[9] In our implementation we do not explicitly compute the storage tree but instead traverse directly the AST provided by the Solidity compiler.

def *packpath (node, subExprs, d, result)***:**
  foreach *expr in subExprs* **do**
    if *expr = id ∨ expr = e.id* **then**
      find edge $node \xrightarrow{id\ (i)} child$;
      $result := result[d \leftarrow i]$;
    if *expr = e[idx]* **then**
      find edge $node \xrightarrow{(i)} child$;
      $result := result[d \leftarrow \mathcal{E}(idx)]$;
    $node, d := child, d + 1$;
  return *result*
def *pack*(*expr*)**:**
  baseExprs := list of base sub-expressions of *expr*;
  baseExpr := car(*baseExprs*);
  if *baseExpr is a state variable* **then**
    return *packpath*(tree(type(*expr*)), *baseExprs*, 0, constarr$_{[int]int}$(*0*))
  if *baseExpr is a storage pointer* **then**
    result := constarr$_{[int]int}$(0);
    prefix := $\mathcal{E}$(*baseExpr*);
    foreach *path to a leaf in* tree(type(*baseExpr*)) **do**
      pathResult, pathCond := *prefix, true*;
      foreach *kth edge on the path with label id (i)* **do**
        $pathCond := pathCond \wedge prefix[k] = i$
      pathResult := packpath(*leaf*, cdr(*baseExprs*), len(*path*), *pathResult*);
      result := *ite*(*pathCond, pathResult, result*);
    return *result*

Fig. 7: Packing of an expressions. It returns a symbolic array expression that, when evaluated, can identify the path to the storage entity that the expression references.

access $e.m_i$ it is recursively the base expressions of e. We call the first element of this list (denoted by car) the base expression (the innermost base expression). The base expression is always either a state variable or a storage pointer, and we consider these two cases separately.

If the *base expression is a state variable*, we simply align the expression along the storage tree with the packpath function. The packpath function takes the list of base sub-expressions, and the storage tree to use for alignment, and then processes the expressions in order. If the current expression is an identifier (state variable or member access), the algorithm finds the outgoing edge annotated with the identifier (from the current node) and writes the index into the result array. If the expression is an index access, the algorithm maps and writes the index expression (symbolically) in the array. The expression mapping function $\mathcal{E}(.)$ is introduced later in Section 3.6.

If the *base expression is a storage pointer*, the process is more general since the "start" of the packing must accommodate any point in storage where the base expression can point to. In this case the algorithm finds all paths to leaves in the

tree of the base pointer, identifies the condition for taking that path and writes the labels on the path to an array. Then it uses packpath to continue writing the array with the rest of the expression (denoted by cdr), as before. Finally, a conditional expression is constructed with all the conditions and packed arrays. Note, that the type of this conditional is still an SMT array of integers as it is the case for a single path.

Example 8. For contract in Figure 6a, pack(ss[8].ts[5]) produces $[2, 8, 1, 5]$ by calling packpath on the base sub-expressions $[ss, ss[8], ss[8].ts, ss[8].ts[5]]$. First, 2 is added as ss is the state variable with index 2. Then, ss[8] is an index access so 8 is mapped to 8 and added to the result. Next, ss[8].ts is a member access with ts having the index 1. Finally, ss[8].ts[5] is an index access so 5 is mapped to 5 and added.

> **def** *unpack*(*ptr*)**:**
> | **return** *unpack*(*ptr*, *tree*(*type*(*ptr*))), *empty*, 0);
> **def** *unpack*(*ptr*, *node*, *expr*, *d*)**:**
> | *result* := *empty*;
> | **if** *node has no outgoing edges* **then** *result* := *expr*;
> | **if** *node is contract* **then**
> | **foreach** *edge node* $\xrightarrow{id\,(i)}$ *child* **do**
> | *result* := ite(*ptr*[*d*] = *i*, unpack(*ptr*, *child*, *id*, *d* + 1), *result*);
> | **if** *node is struct* **then**
> | **foreach** *edge node* $\xrightarrow{id\,(i)}$ *child* **do**
> | *result* := ite(*ptr*[*d*] = *i*, unpack(*ptr*, *child*, *expr*.*id*, *d* + 1), *result*);
> | **if** *node is array/mapping with edge node* $\xrightarrow{(i)}$ *child* **then**
> | *result* := unpack(*ptr*, *child*, *expr*[*ptr*[*d*]], *d* + 1);
> | **return** *result*;

Fig. 8: Unpacking of a local storage pointer into a conditional expression.

Unpacking. The opposite of pack() is unpack(), shown in Figure 8. This function takes a storage pointer (of type $[int]int$) and produces a conditional expression that decodes any given path into one of the leaves of the storage tree. The function recursively traverses the tree starting from the contract node and accumulates the expressions leading to the leaves. The function creates conditionals when branching, and when a leaf is reached the accumulated expression is simply returned. For contracts we process edges corresponding to each state variable by setting the subexpression to be the state variable itself. For structs we process edges corresponding to each member by wrapping the subexpression into a member access. For both contracts and structs, the subexpressions are collected into a conditional as separate cases. For arrays and mappings we process the

single outgoing edge by wrapping the subexpression into an index access using the current element (at index d) of the pointer.

Example 9. For example, the conditional expression corresponding to the tree in Figure 6b can be seen in Figure 6c. Given a pointer *ptr*, if $ptr[0] = 0$ then the conditional evaluates to *t1*. Otherwise, if $ptr[0] = 1$ then *s1* has to be taken, where two leaves are possible: if $ptr[1] = 0$ then the result is *s1.t* otherwise it is $s1.ts[ptr[2]]$, and so on. If *ptr* is $[2, 8, 1, 5]$ then the conditional evaluates exactly to `ss[8].ts[5]` from which *ptr* was packed.[10]

Note that with inheritance and libraries [30] it is possible that a contract defines a type T but has no nodes in its storage tree. The contract can still define functions with storage pointers to T, which can be called by derived contracts that define state variables of type T. In such cases we declare an array of type $[int]\mathcal{T}(T)$, called the *default context*, and unpack storage pointers to T as if the default context was a state variable. This allows us to reason about abstract contracts and libraries, modeling that their storage pointers can point to arbitrary entities not yet declared.

3.3   Contracts, State Variables, Functions

The focus of our discussion is the Solidity memory model and, for presentation purposes, we assume a minimalist setting where the important aspects of storage and memory can be presented: we assume a single contract and a single function to translate. Interactions between multiple functions are handled differently depending on the verification approach. For example, in modular verification functions are checked individually against specifications (pre- and post-conditions) and function calls are replaced by their specification [20].

State variables. Each state variable s_i of a contract is mapped to a variable declaration $s_i : \mathcal{T}(\text{type}(s_i))$ in the SMT program.[11] The data location of state variables is always storage. As discussed previously, reference types are mapped using SMT datatypes and arrays, which ensures non-aliasing by construction. While Solidity optionally allows inline initializer expressions for state variables, without the loss of generality we can assume that they are initialized in the constructor using regular assignments.

---

[10] Note that due to the "else" branches, unpack is a is a non-injective surjective function. For example, $[a, 8, 1, 5]$ with any $a \geq 2$ would evaluate to the same slot. However this does not affect our encoding as pointers cannot be compared and pack always returns the same (unique) values.

[11] Generalizing this to multiple contracts can be done directly by using a separate one-dimensional heap for each state variable, indexed by a receiver parameter (*this* : *address*) identifying the current contract instance (see, e.g., [20]).

$\mathsf{defval}(\texttt{bool}) \quad\; \doteq false$

$\mathsf{defval}(\texttt{address}) \doteq \mathsf{defval}(\texttt{int}) \doteq \mathsf{defval}(\texttt{uint}) \doteq 0$

$\mathsf{defval}(\texttt{mapping(}K\texttt{=>}V\texttt{)}) \doteq \mathsf{constarr}_{[\mathcal{T}(K)]\mathcal{T}(V)}(\mathsf{defval}(V))$

$\mathsf{defval}(T\texttt{[] storage}) \doteq \mathsf{defval}(T\texttt{[0] storage})$

$\mathsf{defval}(T\texttt{[] memory}) \;\doteq \mathsf{defval}(T\texttt{[0] memory})$

$\mathsf{defval}(T\texttt{[}n\texttt{] storage}) \doteq StorArr_T(\mathsf{constarr}_{[int]\mathcal{T}(T)}(\mathsf{defval}(T)), n)$

$\mathsf{defval}(T\texttt{[}n\texttt{] memory}) \;\doteq [ref : int] \text{ (fresh symbol)}$

$$\begin{aligned}
&\{ref := refcnt := refcnt + 1\}\\
&\{arrheap_T[ref].length := n\}\\
&\{arrheap_T[ref].arr[i] := \mathsf{defval}(T)\} \qquad \text{for } 0 \le i \le n\\
&ref
\end{aligned}$$

$\mathsf{defval}(\texttt{struct } S \texttt{ storage}) \doteq StorStruct_S(\dots, \mathsf{defval}(S_i), \dots)$

$\mathsf{defval}(\texttt{struct } S \texttt{ memory}) \;\doteq [ref : int] \text{ (fresh symbol)}$

$$\begin{aligned}
&\{ref := refcnt := refcnt + 1\}\\
&\{structheap_S[ref].m_i = \mathsf{defval}(S_i)\} \text{ for each } m_i\\
&ref
\end{aligned}$$

Fig. 9: Formalization of default values. We denote $\texttt{struct } S$ members as m_i with types S_i.

Functions calls. From the perspective of the memory model, the only important aspect of function calls is the way parameters are passed in and how function return values are treated. Our formalization is general in that it allows us to treat both of the above as plain assignments (explained later in Section 3.5). For each parameter p_i and return value r_i of a function, we add declarations $p_i : \mathcal{T}(\mathsf{type}(p_i))$ and $r_i : \mathcal{T}(\mathsf{type}(r_i))$ in the SMT program. Note that for reference types appearing as parameters or return values of the function, their types are either memory or storage pointers.

Memory allocation. In order to model allocation of new memory entities, while keeping some non-aliasing information, we introduce an allocation counter $refcnt : int$ variable in the preamble of the SMT program. This counter is incremented for each allocation of memory entities and used as the address of the new entity. For each parameter p_i with memory data location we include an assumption $assume(p_i \le refcnt)$ as they can be arbitrary pointers, but should not alias with new allocations within the function. Note that if a parameter of memory pointer type is a reference type containing other references, such non-aliasing constraints need to be assumed recursively [25]. This can be done for structs by enumerating members. But, for dynamic arrays it requires quantification that is nevertheless still decidable (array property fragment [13]).

Initialization and default values. If we are translating the constructor function, each state variable s_i is first initialized to its default value with a statement $s_i := \mathsf{defval}(\mathsf{type}(s_i))$. For regular functions, we set each return value r_i to its default value with a statement $r_i := \mathsf{defval}(\mathsf{type}(r_i))$. We use $\mathsf{defval}(.)$, as defined

in Figure 9, to denote the function that maps a Solidity type to its default value as an SMT expression. Note that, as a side effect, this function can do allocations for memory entities, introducing extra declarations and statements, denoted by [*decl*] and {*stmt*}. As expected, the default value is *false* for Booleans and 0 for other primitives that map to integers. For mappings from K to V, the default value is an SMT constant array returning the default value of the value type V for each key $k \in K$ (see, e.g., [16]). The default value of storage arrays is the corresponding datatype value constructed with a constant array of the default value for base type T, and a length of n or 0 for fixed- or dynamically-sized arrays. For storage structs, the default value is the corresponding datatype value constructed with the default values of each member.

The default value of uninitialized memory pointers is unusual. Since Solidity doesn't support "null" pointers, a new entity is automatically allocated in memory and initialized to default values (which might include additional recursive initialization). Note, that for fixed-size arrays Solidity enforces that the array size n must be an integer literal or a compile time constant, so setting each element to its default value is possible without loops or quantifiers. Similarly for structs, each member is recursively initialized, which is again possible by explicitly enumerating each member.

3.4   Statements

We use $\mathcal{S}[\![.]\!]$ to denote the function that translates Solidity statements to a list of statements in the SMT program. It relies on the type mapping function $\mathcal{T}(.)$ (presented previously in Section 3.1) and on the expression mapping function $\mathcal{E}(.)$ (to be introduced in Section 3.6). Furthermore, we define a helper function $\mathcal{A}(.,.)$ dedicated to modeling Solidity assignments (to be discussed in Section 3.5).

The definition of $\mathcal{S}[\![.]\!]$ is shown in Figure 10. As a side effect, extra declarations can be introduced to the preamble of the SMT program (denoted by [*decl*]). The Solidity documentation [30] does not precisely state the order of evaluating subexpressions in statements. It only specifies that subnodes are processed before the parent node. This problem is independent form the discussion of the memory models so we assume that side effects of subexpressions are added in the same order as it is implemented in the compiler. Furthermore, if a subexpression is mapped multiple times, we assume that the side effects are only added once. This makes our presentation simpler by introducing fewer temporary variables.

Local variable declarations introduce a variable declaration with the same identifier in the SMT program by mapping the type.[12] If an initialization expression is given, it is mapped using $\mathcal{E}(.)$ and assigned to the variable. Otherwise, the default value is used as defined by defval(.) in Figure 9. Delete assigns the default value for a type, which is simply mapped to an assignment in our formalization. Solidity supports multiple assignments as one statement with a tuple-like syntax. The documentation [30] does not specify the behavior precisely, but the

---

[12] Without the loss of generality we assume that identifiers in Solidity are unique. The compiler handles scoping and assigns an unique identifier to each declaration.

$$\mathcal{S}[\![T\ id]\!] \doteq [id : \mathcal{T}(T)]; \mathcal{A}(id, \mathsf{defval}(T))$$
$$\mathcal{S}[\![T\ id = expr]\!] \doteq [id : \mathcal{T}(T)]; \mathcal{A}(id, \mathcal{E}(expr))$$
$$\mathcal{S}[\![\texttt{delete}\ e]\!] \doteq \mathcal{A}(\mathcal{E}(e), \mathsf{defval}(\mathsf{type}(e)))$$

$$\mathcal{S}[\![l_1, \ldots, l_n\ =\ r_1, \ldots, r_n]\!] \doteq [tmp_i : \mathcal{T}(\mathsf{type}(r_i))] \text{ for } 1 \leq i \leq n \text{ (fresh symbols)}$$
$$\mathcal{A}(tmp_i, \mathcal{E}(r_i)) \qquad \text{for } 1 \leq i \leq n$$
$$\mathcal{A}(\mathcal{E}(l_i), tmp_i) \qquad \text{for } n \geq i \geq 1 \text{ (reversed)}$$

$$\mathcal{S}[\![e_1.\texttt{push}(e_2)]\!] \doteq \mathcal{A}(\mathcal{E}(e_1).arr[\mathcal{E}(e_1).length], \mathcal{E}(e_2))$$
$$\mathcal{E}(e_1).length := \mathcal{E}(e_1).length + 1$$
$$\mathcal{S}[\![e.\texttt{pop}()]\!] \doteq \mathcal{E}(e).length := \mathcal{E}(e).length - 1$$
$$\mathcal{A}(\mathcal{E}(e).arr[\mathcal{E}(e).length], \mathsf{defval}(\mathsf{arrtype}(\mathcal{E}(e))))$$

Fig. 10: Formalization of statements.

```
contract C {
  struct S { int x; }

  S s1, s2, s3;

  function primitiveAssign() {
    s1.x = 1; s2.x = 2; s3.x = 3;
    (s1.x, s3.x, s2.x) = (s3.x, s2.x, s1.x);
    // s1.x == 3, s2.x == 1, s3.x == 2
  }
  function storageAssign() {
    s1.x = 1; s2.x = 2; s3.x = 3;
    (s1, s3, s2) = (s3, s2, s1);
    // s1.x, s2.x, s3.x are all equal to 1
  }
}
```

Fig. 11: Example illustrating the right-to-left assignment order and the treatment of reference types in storage in tuple assignment.

```
contract C {
  struct S { int x; }

  S[] a;

  constructor() {
    a.push(S(1));
    S storage s = a[0];
    a.pop();
    assert(s.x == 1); // Ok
    // Following is error
    // assert(a[0].x == 1);
  }
}
```

Fig. 12: Example illustrating a dangling pointer to storage.

compiler first evaluates the RHS and LHS tuples (in this order) from left to right and then assignment is performed component-wise from right to left.

Example 10. Consider the tuple assignment in function `primitiveAssign()` in Figure 11. From right to left, `s2.x` is assigned first with the value of `s1.x` which is 1. Afterwards, when `s3.x` is assigned with `s2.x`, the already evaluated (old) value of 2 is used instead of the new value 1. Finally, `s1.x` gets the old value of `s3.x`, i.e., 3. Note however, that storage expressions on the RHS evaluate to storage pointers. Consider, for example, the function `storageAssign()` in Figure 11. From right to left, `s2` is assigned first, with a pointer to `s1` making `s2.x` become 1. However, as opposed to primitive types, when `s3` is assigned next, `s2` on the RHS is a storage pointer and thus the new value in the storage of `s2` is assigned to `s3` making `s3.x` become 1. Similarly, `s1.x` also becomes 1 as the new value behind `s3` is used.

Array push increases the length and assigns the given expression as the last element. Array pop decreases the length and sets the removed element to its default value. While the removed element can no longer be accessed via indexing into an array (a runtime error occurs), it can still be accessed via local storage pointers (see Figure 12).[13]

3.5   Assignments

Assignments between reference types in Solidity can be either pointer assignments or value assignments, involving deep copying and possible new allocations in the latter case. We use $\mathcal{A}(lhs, rhs)$ to denote the function that assigns a *rhs* SMT expression to a *lhs* SMT expression based on their original types and data locations. The definition of $\mathcal{A}(.,.)$ is shown in Figure 13. Value type assignments are simply mapped to an SMT assignment. To make our presentation more clear, we subdivide the other cases into separate functions for array, struct and mapping operands, denoted by $\mathcal{A}_A(.,.)$, $\mathcal{A}_S(.,.)$ and $\mathcal{A}_M(.,.)$ respectively.

Mappings. As discussed previously, Solidity prohibits direct assignment of mappings. However, it is possible to declare a storage pointer to a mapping, in which case the RHS expression is packed. It is also possible to assign two storage pointers, which simply assigns pointers. Other cases are a no-op.[14]

Structs and arrays. For structs and arrays the semantics of assignment is summarized in Figure 14. However, there are some notable details in various cases that we expand on below.

Assigning anything *to storage* LHS always causes a deep copy. If the RHS is storage, this is simply mapped to a datatype assignment in our encoding (with an additional unpacking if the RHS is storage pointer).[15] If the RHS is memory, deep copy for structs can be done member wise by accessing the heap with the RHS pointer and performing the assignment recursively (as members can be reference types themselves). For arrays, we access the datatype corresponding to the array via the heap and do an assignment, which does a deep copy in SMT. Note however, that this only works if the base type of the array is a value type. For reference types, memory array elements are pointers and would require being dereferenced during assignment to storage. As opposed to struct members, the number of array elements is not known at compile time so loops or quantifiers have to be used (as in traditional software analysis). However, this is a

---

[13] The current version (0.5.x) of Solidity supports resizing arrays by assigning to the length member. However, this behavior is dangerous and has been since removed in the next version (0.6.0) (see https://solidity.readthedocs.io/en/v0.6.0/060-breaking-changes.html). Therefore, we do not support this in our encoding.

[14] This is consequence of the fact that keys are not stored in mappings and so the assignment is impossible to perform.

[15] This also causes mappings to be copied, which contradicts the current semantics. However, we chose to keep the deep copy as assignments of mappings is planned to be disallowed in the future (see https://github.com/ethereum/solidity/issues/7739).

$$\mathcal{A}(lhs, rhs) \doteq lhs := rhs \quad \text{for value type operands}$$
$$\mathcal{A}(lhs, rhs) \doteq \mathcal{A}_M(lhs, rhs) \text{ for mapping type operands}$$
$$\mathcal{A}(lhs, rhs) \doteq \mathcal{A}_S(lhs, rhs) \text{ for struct type operands}$$
$$\mathcal{A}(lhs, rhs) \doteq \mathcal{A}_A(lhs, rhs) \text{ for array type operands}$$

$$\mathcal{A}_M(lhs : \texttt{sp}, rhs : \texttt{s}) \doteq lhs := \mathsf{pack}(rhs)$$
$$\mathcal{A}_M(lhs : \texttt{sp}, rhs : \texttt{sp}) \doteq lhs := rhs$$
$$\mathcal{A}_M(lhs, rhs) \qquad\quad \doteq \{\} \qquad\qquad (\text{all other cases})$$

$$\mathcal{A}_S(lhs : \texttt{s}, rhs : \texttt{s}) \quad \doteq lhs := rhs$$
$$\mathcal{A}_S(lhs : \texttt{s}, rhs : \texttt{m}) \quad \doteq \mathcal{A}(lhs.m_i, structheap_{\mathsf{type}(rhs)}[rhs].m_i) \text{ for each } m_i$$
$$\mathcal{A}_S(lhs : \texttt{s}, rhs : \texttt{sp}) \quad \doteq \mathcal{A}_S(lhs, \mathsf{unpack}(rhs))$$
$$\mathcal{A}_S(lhs : \texttt{m}, rhs : \texttt{m}) \quad \doteq lhs := rhs$$
$$\mathcal{A}_S(lhs : \texttt{m}, rhs : \texttt{s}) \quad \doteq lhs := refcnt := refcnt + 1$$
$$\qquad\qquad\qquad\qquad\qquad \mathcal{A}(structheap_{\mathsf{type}(lhs)}[lhs].m_i, rhs.m_i) \text{ for each } m_i$$
$$\mathcal{A}_S(lhs : \texttt{m}, rhs : \texttt{sp}) \quad \doteq \mathcal{A}_S(lhs, \mathsf{unpack}(rhs))$$
$$\mathcal{A}_S(lhs : \texttt{sp}, rhs : \texttt{s}) \quad \doteq lhs := \mathsf{pack}(rhs)$$
$$\mathcal{A}_S(lhs : \texttt{sp}, rhs : \texttt{sp}) \doteq lhs := rhs$$

$$\mathcal{A}_A(lhs : \texttt{s}, rhs : \texttt{s}) \quad \doteq lhs := rhs$$
$$\mathcal{A}_A(lhs : \texttt{s}, rhs : \texttt{m}) \quad \doteq lhs := arrheap_{\mathsf{type}(rhs)}[rhs]$$
$$\mathcal{A}_A(lhs : \texttt{s}, rhs : \texttt{sp}) \quad \doteq \mathcal{A}_A(lhs, \mathsf{unpack}(rhs))$$
$$\mathcal{A}_A(lhs : \texttt{m}, rhs : \texttt{m}) \quad \doteq lhs := rhs$$
$$\mathcal{A}_A(lhs : \texttt{m}, rhs : \texttt{s}) \quad \doteq lhs := refcnt := refcnt + 1$$
$$\qquad\qquad\qquad\qquad\qquad arrheap_{\mathsf{type}(lhs)}[lhs] := rhs$$
$$\mathcal{A}_A(lhs : \texttt{m}, rhs : \texttt{sp}) \quad \doteq \mathcal{A}_A(lhs, \mathsf{unpack}(rhs))$$
$$\mathcal{A}_A(lhs : \texttt{sp}, rhs : \texttt{s}) \quad \doteq lhs := \mathsf{pack}(rhs)$$
$$\mathcal{A}_A(lhs : \texttt{sp}, rhs : \texttt{sp}) \doteq lhs := rhs$$

Fig. 13: Formalization of assignment based on different type categories and data locations for the LHS and RHS. We use $\texttt{s}$, $\texttt{sp}$ and $\texttt{m}$ after the arguments to denote storage, storage pointer and memory types respectively.

special case, which can be encoded in the decidable array property fragment [13]. Assigning storage (or storage pointer) *to memory* is also a deep copy but in the other direction. However, instead overwriting the existing memory entity, a new one is allocated (recursively for reference typed elements or members). We model this by incrementing the reference counter, storing it in the LHS and then accessing the heap for deep copy using the new pointer.

3.6   Expressions

We use $\mathcal{E}(.)$ to denote the function that translates a Solidity expression to an SMT expression. As a side effect, declarations and statements might be introduced (denoted by $[decl]$ and $\{stmt\}$ respectively). The definition of $\mathcal{E}(.)$ is shown in Figure 15. As discussed in Section 3.4 we assume that side effects are added from subexpressions in the proper order and only once.

Member access is mapped to an SMT member access by mapping the base expression and the member name. There is an extra unpacking step for storage

lhs/rhs	Storage	Memory	Stor.ptr.
Storage	Deep copy	Deep copy	Deep copy
Memory	Deep copy	Pointer assign	Deep copy
Stor.ptr.	Pointer assign	Error	Pointer assign

Fig. 14: Semantics of assignment between array and struct operands based on their data location.

$$\mathcal{E}(id) \doteq id$$

$$
\begin{array}{lll}
\mathcal{E}(expr.id) \doteq \mathcal{E}(expr).\mathcal{E}(id) & \text{if } \mathsf{type}(expr) = \mathtt{struct}\ S\ \mathtt{storage} \\
\mathcal{E}(expr.id) \doteq \mathsf{unpack}(\mathcal{E}(expr)).\mathcal{E}(id) & \text{if } \mathsf{type}(expr) = \mathtt{struct}\ S\ \mathtt{storptr} \\
\mathcal{E}(expr.id) \doteq structheap_S[\mathcal{E}(expr)].\mathcal{E}(id) & \text{if } \mathsf{type}(expr) = \mathtt{struct}\ S\ \mathtt{memory} \\
\mathcal{E}(expr.id) \doteq \mathcal{E}(expr).\mathcal{E}(id) & \text{if } \mathsf{type}(expr) = T\mathtt{[]}\ \mathtt{storage} \\
\mathcal{E}(expr.id) \doteq \mathsf{unpack}(\mathcal{E}(expr)).\mathcal{E}(id) & \text{if } \mathsf{type}(expr) = T\mathtt{[]}\ \mathtt{storptr} \\
\mathcal{E}(expr.id) \doteq arrheap_T[\mathcal{E}(expr)].\mathcal{E}(id) & \text{if } \mathsf{type}(expr) = T\mathtt{[]}\ \mathtt{memory}
\end{array}
$$

$$
\begin{array}{lll}
\mathcal{E}(expr[idx]) \doteq \mathcal{E}(expr).arr[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = T\mathtt{[]}\ \mathtt{storage} \\
\mathcal{E}(expr[idx]) \doteq \mathsf{unpack}(\mathcal{E}(expr)).arr[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = T\mathtt{[]}\ \mathtt{storptr} \\
\mathcal{E}(expr[idx]) \doteq arrheap_T[\mathcal{E}(expr)].arr[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = T\mathtt{[]}\ \mathtt{memory} \\
\mathcal{E}(expr[idx]) \doteq \mathcal{E}(expr)[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = \mathtt{mapping(}K\mathtt{=>}V\mathtt{)}\ \mathtt{storage} \\
\mathcal{E}(expr[idx]) \doteq \mathsf{unpack}(\mathcal{E}(expr))[\mathcal{E}(idx)] & \text{if } \mathsf{type}(expr) = \mathtt{mapping(}K\mathtt{=>}V\mathtt{)}\ \mathtt{storptr}
\end{array}
$$

$$
\begin{aligned}
\mathcal{E}(cond\ \mathtt{?}\ expr_T\ \mathtt{:}\ expr_F) \doteq\ & [var_T : \mathcal{T}(\mathsf{type}(cond\ \mathtt{?}\ expr_T\ \mathtt{:}\ expr_F))]\ (\text{fresh symbol}) \\
& [var_F : \mathcal{T}(\mathsf{type}(cond\ \mathtt{?}\ expr_T\ \mathtt{:}\ expr_F))]\ (\text{fresh symbol}) \\
& \{\mathcal{A}(var_T, \mathcal{E}(expr_T))\} \\
& \{\mathcal{A}(var_F, \mathcal{E}(expr_F))\} \\
& ite(\mathcal{E}(cond), var_T, var_F)
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}(\mathtt{new}\ T\mathtt{[]}(expr)) \doteq\ & [ref : int]\ (\text{fresh symbol}) \\
& \{ref := refcnt := refcnt + 1\} \\
& \{arrheap_T[ref].length := \mathcal{E}(expr)\} \\
& \{arrheap_T[ref].arr[i] := \mathsf{defval}(T)\}\ \text{for } 0 \le i \le \mathcal{E}(expr) \\
& ref
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}(S(\ldots, expr_i, \ldots)) \doteq\ & [ref : int]\ (\text{fresh symbol}) \\
& \{ref := refcnt := refcnt + 1\} \\
& \{structheap_S[ref].m_i := \mathcal{E}(expr_i)\}\ \text{for each member } m_i \\
& ref
\end{aligned}
$$

Fig. 15: Formalization of expressions. We denote `struct` S members as m_i with types S_i.

pointers and a heap access for memory. Note that the only valid member for arrays is `length`. Index access is mapped to an SMT array read by mapping the base expression and the index, and adding en extra member access for arrays to get the inner array *arr* of elements from the datatype. Furthermore, similarly to member accesses, an extra unpacking step is needed for storage pointers and a heap access for memory.

Conditionals in Solidity can be mapped to an SMT conditional in general. However, data locations can be different for the true and false branches, causing possible side effects. Therefore, we first introduce fresh variables for the true and false branch with the common type (of the whole conditional), then make assignments using $\mathcal{A}(.,.)$ and finally use the new variables in the conditional. The documentation [30] does not specify the common type, but the compiler returns memory if any of the branches is memory, and storage pointer otherwise.

Allocating a new array in memory increments the reference counter, sets the length and the default values for each element (recursively). Note that in general the length might not be a compile time constant,in which case setting default values could be encoded with the array property fragment (similarly to deep copy in assignments) [13]. Allocating a new memory struct also increments the reference counter and sets each value by translating the provided arguments.

4   Evaluation

The formalization described in this paper serves as the basis of our Solidity verification tool SOLC-VERIFY [20].[16] In this section we provide an evaluation of the presented formalization and our implementation by validating it on a set of relevant test cases. For illustrative purposes we also compare our tool with other available Solidity analysis tools.[17]

"Real world" contracts currently deployed on Ethereum (e.g., contract available on Etherscan) have limited value for evaluating memory model semantics. Many such contracts use old compiler versions with constructs that are not supported anymore, and do not use newer features. There are also many toy and trivial contracts that are deployed but not used, and popular contracts (e.g. tokens) are over-represented with many duplicates. Furthermore, the inconsistent usage of `assert` and `require` [20] makes evaluation hard. Evaluating the memory semantics requires contracts that exercise diverse features of the memory model. There are larger dApps that do use more complex features (e.g., Augur or ENS), but these contracts also depend on many other features (e.g. inheritance, modifiers, loops) that would skew the results.

Therefore we have manually developed a set of tests that try to capture the interesting behaviors and corner cases of the Solidity memory semantics. The tests are targeted examples that do not use irrelevant features. The set is structured so that every target test behavior is represented with a test case that sets up the state, exercises a specific feature and checks the correctness of the behavior with assertions. This way a test should only pass if the tool provides a correct verification result by modeling the targeted feature precisely.

---

[16] SOLC-VERIFY is open source, available at https://github.com/SRI-CSL/solidity. Besides certain low-level constructs (such as inline assembly) SOLC-VERIFY supports a majority of Solidity features that we omitted from the presentation, including inheritance, function modifiers, for/while loops and if-then-else.

[17] All tests, with a Truffle test harness, a docker container with all the tools, and all individual results are available at https://github.com/dddejan/solidity-semantics-tests.

The correctness of the tests themselves is determined by running them through the EVM with no assertion failures. Test cases are expanded to use all reference types and combinations of reference types. This includes structures, mappings, dynamic and fixed-size arrays, both single- and multi-dimensional.

The tests are organized into the following classes. Tests in the `assignment` class check whether the assign statement is properly modeled. This includes assignments in the same data location, but also assignments across data locations that need deep copying, and assignments and re-assignments of memory and storage pointers. The `delete` class of tests checks whether the `delete` statement is properly modeled. Tests in the `init` class check whether variable and data initialization is properly modeled. For variables in storage, we check if they are properly initialized to default values in the contract constructor. Similarly, we check whether memory variables are properly initialized to provided values, or default values when no initializer is provided. The `storage` class of tests checks whether storage itself is properly modeled for various reference types, including for example non-aliasing. Tests in the `storageptr` class check whether storage pointers are modeled properly. This includes checking if the model properly treats storage pointers to various reference types, including nested types. In addition, the tests check that the storage pointers can be properly passed to functions and ensure non-aliasing for distinct parts of storage.

For illustrative purposes we include a comparison with the following available Solidity analysis tools: MYTHRIL v0.21.17 [29], VERISOL v0.1.1-alpha [24], and SMT-CHECKER v0.5.12 [1]. MYTHRIL is a Solidity symbolic execution tool that runs analysis at the level of the EVM bytecode. VERISOL is similar to SOLC-VERIFY in that it uses Boogie to model the Solidity contracts, but takes the traditional approach to modeling memory and storage with pointers and quantifiers. SMT-CHECKER is an SMT-based analysis module built into the Solidity compiler itself. There are other tools that can be found in the literature, but they are either basic prototypes that cannot handle realistic features we are considering, or are not available for direct comparison.

We ran the experiments on a machine with Intel Xeon E5-4627 v2 @ 3.30GHz CPU enforcing a 60s timeout and a memory limit of 64GB. Results are shown in Table 1. As expected, MYTHRIL has the most consistent results on our test set. This is because MYTHRIL models contract semantics at the EVM level and does not need to model complex Solidity semantics. Nevertheless, the results also indicate that the performance penalty for this precision is significant (8 timeouts). VERISOL, as the closest to our approach, still doesn't support many features and has a significant amount of false reports for features that it does support. Many false reports are because their model of storage is based on pointers and tries to ensure storage consistency with the use of quantifiers. SMT-CHECKER doesn't yet support the majority of the Solidity features that our tests target.

Based on the results, SOLC-VERIFY performs well on our test set, matching the precision of MYTHRIL at very low computational cost. The few false alarms we have are either due to Solidity features that we chose to not implement (e.g., proper treatment of `mapping` assignments), or parts of the semantics that we

Table 1: Results of evaluating MYTHRIL, VERISOL, SMT-CHECKER, and SOLC-VERIFY on our test suite.

assignment (102)	correct	incorrect	unsupported	timeout	time (s)
MYTHRIL	94	0	0	8	1655.14
VERISOL	10	61	31	0	175.27
SMT-CHECKER	6	9	87	0	15.25
SOLC-VERIFY	78	8	16	0	62.81
delete (14)	correct	incorrect	unsupported	timeout	time (s)
MYTHRIL	13	1	0	0	47.51
VERISOL	3	8	3	0	24.66
SMT-CHECKER	0	0	14	0	0.30
SOLC-VERIFY	7	1	6	0	9.02
init (18)	correct	incorrect	unsupported	timeout	time (s)
MYTHRIL	15	3	0	0	59.67
VERISOL	7	8	3	0	28.82
SMT-CHECKER	0	0	18	0	0.41
SOLC-VERIFY	13	5	0	0	11.88
storage (27)	correct	incorrect	unsupported	timeout	time (s)
MYTHRIL	27	0	0	0	310.40
VERISOL	12	15	0	0	43.45
SMT-CHECKER	2	0	25	0	1.32
SOLC-VERIFY	27	0	0	0	17.61
storageptr (164)	correct	incorrect	unsupported	timeout	time (s)
MYTHRIL	164	0	0	0	1520.29
VERISOL	128	19	17	0	203.93
SMT-CHECKER	4	18	142	0	21.93
SOLC-VERIFY	164	0	0	0	96.92

only implemented partially (such as deep copy of arrays with reference types and recursively initializing memory objects). There are no technical difficulties in supporting them and they are planned in the future.

5   Related Work

There is a strong push in the Ethereum community to apply formal methods to smart contract verification. This includes many attempts to formalize the semantics of smart contracts, both at the level of EVM and Solidity.

EVM-level semantics. Bhargavan et al. [11] decompile a fragment of EVM to F*, modeling EVM as a stack based machine with word and byte arrays for storage and memory. Grishchenko et al. [19] extend this work by providing a small step semantics for EVM. KEVM [21] provides an executable formal semantics of EVM in the K framework. Hirai [22] formalizes EVM in Lem, a language used by

some interactive theorem provers. Amani et al. [2] extends this work by defining a program logic to reason about EVM bytecode.

Solidity-level semantics. Jiao et al. [23] formalize the operational semantics of Solidity in the K framework. Their formalization focuses on the details of bit-precise sizes of types, alignment and padding in storage. They encode storage slots, arrays and mappings with the full encoding of hashing. However, the formalization does not describe assignments (e.g., deep copy) apart from simple cases. Furthermore, user defined structs are also not mentioned. In contrast, our semantics is high-level and abstracts away some details (e.g., hashes, alignments) to enable efficient verification. Additionally, we provide proper modeling of different cases for assignments between storage and memory. Bartotelli et al. [10] propose TINYSOL, a minimal core calculus for a subset of Solidity, required to model basic features such as asset transfer and reentrancy. Contract data is modeled as a key value store, with no differences in storage and memory, or in value and reference types. Crafa et al. [15] introduce Featherweight Solidity, a calculus formalizing core features of the language, with focus on primitive types. Data locations and reference types are not discussed, only mappings are mentioned briefly. The main focus is on the type system and type checking. They propose an improved type system that can statically detect unsafe casts and callbacks. The closest to our work is the work of Zakrzewski [33], a Coq formalization focusing on functions, modifiers, and the memory model. The memory model is treated similarly: storage is a mapping from names to storage objects (values), memory is a mapping from references to memory objects (containing references recursively) and storage pointers define a path in storage. Their formalization is also high-level, without considering alignment, padding or hashing. The formalization is provided as big step functional semantics in Coq. While the paper presents some example rules, the formalization does not cover all cases. For example the details of assignments (e.g., memory to storage), push/pop for arrays, treating memory aliasing and new expressions. Furthermore, our approach focuses on SMT and modular verification, which enables automated reasoning.

6   Conclusion

We presented a high-level SMT-based formalization of the Solidity memory model semantics. Our formalization covers all aspects of the language related to managing both the persistent contract storage and the transient local memory. The novel encoding of storage pointers as arrays allows us to precisely model non-aliasing and deep copy assignments between storage entities without the need for quantifiers. The memory model forms the basis of our Solidity-level modular verification tool SOLC-VERIFY. We developed a suite of test cases exercising all aspects of memory management with different combinations of reference types. Results indicate that our memory model outperforms existing Solidity-level tools in terms of soundness and precision, and is on par with low-level EVM-based implementations, while having a significantly lower computational cost for discharging verification conditions.

References

1. Alt, L., Reitwiessner, C.: SMT-based verification of Solidity smart contracts. In: ISoLA 2018, LNCS, vol. 11247, pp. 376–388. Springer (2018). https://doi.org/10.1007/978-3-030-03427-6_28
2. 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. ACM (2018)
3. Antonopoulos, A., Wood, G.: Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly Media, Inc. (2018)
4. Atzei, N., Bartoletti, M., Cimoli, T.: A survey of attacks on Ethereum smart contracts. In: POST 2017, LNCS, vol. 10204, pp. 164–186. Springer (2017). https://doi.org/10.1007/978-3-662-54455-6_8
5. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: FMCO 2005, LNCS, vol. 4111, pp. 364–387. Springer (2006). https://doi.org/10.1007/11804192_17
6. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV 2011, LNCS, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14
7. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2016), www.SMT-LIB.org
8. Barrett, C., Shikanian, I., Tinelli, C.: An abstract decision procedure for satisfiability in the theory of recursive data types. Journal on Satisfiability, Boolean Modeling and Computation **3**, 21–46 (2007)
9. Barrett, C., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Model Checking, pp. 305–343. Springer (2018)
10. Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for Solidity contracts. In: DPM 2019, CBT 2019, LNCS, vol. 11737, pp. 233–243. Springer (2019). https://doi.org/978-3-030-31500-9_15
11. 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. pp. 91–96. ACM (2016)
12. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability. IOS press (2009)
13. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: VMCAI 2006, LNCS, vol. 3855, pp. 427–442. Springer (2006). https://doi.org/10.1007/11609773_28
14. Chen, H., Pendleton, M., Njilla, L., Xu, S.: A survey on ethereum systems security: Vulnerabilities, attacks and defenses (2019), https://arxiv.org/abs/1908.04507
15. Crafa, S., Pirro, M.D., Zucca, E.: Is solidity solid enough? In: Financial Cryptography Workshops (2019)
16. De Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: Formal Methods in Computer-Aided Design. pp. 45–52. IEEE (2009)
17. Dhillon, V., Metcalf, D., Hooper, M.: The DAO hacked. In: Blockchain Enabled Applications, pp. 67–78. Apress (2017)
18. Filliâtre, J.C., Paskevich, A.: Why3 — where programs meet provers. In: ESOP 2013, LNCS, vol. 7792, pp. 125–128. Springer (2013). https://doi.org/10.1007/978-3-642-37036-6_8
19. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of Ethereum smart contracts. In: POST 2018, LNCS, vol. 10804, pp. 243–269. Springer (2018). https://doi.org/10.1007/978-3-319-89722-6_10

20. Hajdu, Á., Jovanović, D.: SOLC-VERIFY: A modular verifier for Solidity smart contracts. In: VSTTE 2019, LNCS, vol. 12301. Springer (2019), (In press)
21. 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., IDEALS (2017)
22. Hirai, Y.: Defining the Ethereum virtual machine for interactive theorem provers. In: FC 2017, LNCS, vol. 10323, pp. 520–535. Springer (2017). https://doi.org/10.1007/978-3-319-70278-0_33
23. Jiao, J., Kan, S., Lin, S., Sanán, D., Liu, Y., Sun, J.: Executable operational semantics of Solidity (2018), http://arxiv.org/abs/1804.01295
24. Lahiri, S.K., Chen, S., Wang, Y., Dillig, I.: Formal specification and verification of smart contracts for azure blockchain. In: VSTTE 2019, LNCS, vol. 12301. Springer, (In press)
25. Leino, K.R.M.: Ecstatic: An object-oriented programming language with an axiomatic semantics. In: Proceedings of the Fourth International Workshop on Foundations of Object-Oriented Languages (1997)
26. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR 2010, LNCS, vol. 11247, pp. 348–370. Springer (2010). https://doi.org/10.1007/978-3-642-17511-4_20
27. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress. pp. 21–28 (1962)
28. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS 2008, LNCS, vol. 4963, pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
29. Mueller, B.: Smashing Ethereum smart contracts for fun and real profit. In: Proceedings of the 9th Annual HITB Security Conference (HITBSecConf) (2018)
30. Solidity documentation (2019), https://solidity.readthedocs.io/
31. Szabo, N.: Smart contracts (1994)
32. Wood, G.: Ethereum: A secure decentralised generalised transaction ledger (2017), https://ethereum.github.io/yellowpaper/paper.pdf
33. Zakrzewski, J.: Towards verification of Ethereum smart contracts: A formalization of core of Solidity. In: VSTTE 2018, LNCS, vol. 11294, pp. 229–247. Springer (2018). https://doi.org/10.1007/978-3-030-03592-1_13