# Type Safety for Isabelle/Solidity

Billy Thornton[1][0000−0001−7242−5890] and Diego Marmsoler[1][0000−0003−2859−7673]

University of Exeter, Department of Computer Science, The Innovation Centre, Exeter, EX4 4RN, United Kingdom

**Abstract** Smart contracts are programs stored on the blockchain that are often developed in a high-level programming language, the most popular of which is Solidity. Smart contracts are used to automate financial transactions; thus, bugs can lead to large financial losses. In previous works, we developed a formalization of Solidity in Isabelle/HOL, which can be used to verify the properties of Solidity smart contracts. This formalization features untyped stores, where the types are inferred using a type environment. It is currently unclear whether our semantics can cause type inconsistencies between stores and their environments. With this work we address this problem by formalizing the notion of type safety for untyped stores and verifying that our semantics preserve this property. The formalization and proofs were verified using Isabelle/HOL. Our results guarantee that our semantics are type safe and the proofs can be used to simplify the verification process for the properties of Solidity smart contracts.

**Keywords:** Type Safety · Smart Contracts · Solidity · Program Verification · Isabelle/Solidity.

## 1 Introduction

Blockchain [23] is a novel technology for providing decentralised and openly accessible ledgers without the need of a trusted third party. Originally designed to support cryptocurrencies, blockchain have been used in a wide range of applications, such as *finance* [17], *healthcare* [5], *land management* [10], *business process management* [22], and even *identity management* [30].

Blockchain ecosystems have been rapidly evolving, with *smart contracts* being one of the primary innovations. Smart contracts are digital contracts used to automate transactions on the blockchain once pre-defined conditions are met. For instance, a payment for an item might be released instantly once the buyer and seller have met all specified parameters for a deal. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions [29].

Smart contracts consist of non-modifiable code which is deployed to the blockchain and which can be executed by sending a special transaction to it. Smart contracts are usually developed in a high-level programming language, the most popular of which is *Solidity* [14]. Currently, 90% of all smart contracts are

developed using Solidity [18] and, according to a 2023 survey, Solidity is by far the most popular language used by blockchain developers [4].

As with every computer program, smart contracts may contain bugs which can be exploited. However, since smart contracts are often used to automate financial transactions, such exploits may result in huge economic losses. In general, it is estimated that since 2019, more than $5B was stolen due to vulnerabilities in smart contracts [11].

The high impact of vulnerabilities in smart contracts, together with the fact that once deployed to the blockchain, they cannot be updated or removed easily, makes it important to "get them right" before they are deployed. Consequently, there has been a growing amount of work to verify smart contracts (see [3] for an overview). There is, however, a lack of work which focuses on the analysis of Solidity and in particular its type safety.

Thus, with the following work we formalize and verify type safety for Isabelle/Solidity [19,20,21], a formalization of Solidity in Isabelle [24]. To this end, we provide the following contributions:

- We provide a formal definition of type safety in the context of Isabelle/Solidity (section 3). The definition highlights different aspects of type safety in our context, such as type consistency, structural consistency, and contract properties.
- We verify that the semantics of Isabelle/Solidity preserves type safety (section 4). In particular the verification of pointer structures available in Solidity was challenging and is discussed in more detail.

All of our work is mechanized in Isabelle and the formalization and verification of type safety consists of around 7 000 lines of Isabelle/Isar code.

In the following, we first provide a brief introduction into Isabelle/Solidity in section 2. Then, in section 3 we discuss our definition of type safety and in section 4 its verification. We then discuss related work in section 5 and conclude the paper with a discussion of results in section 6.

## 2 Isabelle/Solidity

Our formalisation of the type safety of Solidity is based on the denotational semantics of Solidity described in [19,20,21] which we summarize in this section. Isabelle/Solidity is developed using higher-order logic with inductive data types [7]. To this end, we use **bold** font for types and *italics* for type constructors. We shall also use

$$\mathbf{type}_\perp \stackrel{\text{def}}{=} \perp \cup \{x_\perp \mid x \in \mathbf{type}\}$$

to denote the type which adds a distinct element $\perp$ to the elements of **type**. Sometimes we shall use

$$\mathbf{type} \rightharpoonup \mathbf{type} \stackrel{\text{def}}{=} \mathbf{type} \to \mathbf{type}_\perp$$

to denote the type of partial functions. For such a function $f$ we shall use $dom(f)$ to denote its domain.

### 2.1 Value Types

Our version of Solidity supports four basic data types, called *value types*:

$$\textbf{Types} \quad ::= \quad \mathit{TBool} \mid \mathit{TAddr} \mid \mathit{TSInt}(\textbf{Nat}) \mid \mathit{TUInt}(\textbf{Nat})$$

*TBool* denotes boolean values and *TAddr* denotes addresses. We also support *signed* and *unsigned* integers from 8 to 256 bits in steps of 8. Thus, $\mathit{TSInt}(b)$ and $\mathit{TUInt}(b)$ denote signed and unsigned integers of bit size $b$. In Solidity, raw data is encoded and stored in hexadecimal format, however, to simplify the computation of locations for reference types, we use the string representation of values for raw data in our model. Thus, type **Valuetype** is actually just a synonym for type **String**, and it is used to represent the data of value types in the store. In addition, we write $\lfloor v \rfloor$ and $\lceil v \rceil$ to convert the value $v$ of a basic data type to and from a string representation, respectively.

### 2.2 Stores and Reference Types

We use strings to model the addresses of storage cells to simplify the computation of locations for reference types. Thus, type **Loc** denotes these types of strings and is used to represent storage locations. We can then model a general store for values of type $v$ as a parametric data type:

$$\textbf{Store}(v) \stackrel{\text{def}}{=} (\textbf{Loc} \rightharpoonup v) \times \textbf{Nat}$$

A store consists of a (finite) mapping to assign values of type $v$ to locations, and in addition, it holds a pointer to the next free location. In the following we denote the mapping of a store $s$ with $\mathit{mapping}(s)$ and we use $\mathit{toploc}(s)$ to denote its top location. We shall also use functions

$$\begin{aligned} &\mathit{accessStore}(l, s) \stackrel{\text{def}}{=} \mathit{mapping}(s)(l) \\ &\mathit{updateStore}(l, \mathit{val}, s) \stackrel{\text{def}}{=} s[\mathit{mapping} \mapsto \mathit{mapping}(s)[l \mapsto \mathit{val}]] \end{aligned}$$

to access and update a location $l$ of a store $s$.

**Computing storage locations** Solidity computes storage locations for reference types by combining the address of the reference type variable with the corresponding index and hashing the result using the Keccak hash function [8]. In our model, the location of the storage cell which holds the value of an element *ix* of a reference type which is stored at location *loc* is obtained by concatenating *ix* with *loc* separated by a dot:

$$h(\mathit{loc},\ \mathit{ix}) \stackrel{\text{def}}{=} \mathit{ix} + \text{“.”} + \mathit{loc} \tag{1}$$

**Stack** The *stack* stores the values for variables which can either be concrete values (for value type variables) or pointers to either memory, calldata, or storage (for reference type variables). Thus, a stack can be modelled as a store which can contain four different types of values:

$$\begin{aligned} \textbf{Stackvalue} &::= \mathit{Simple}(\textbf{Valuetype}) \mid \mathit{Memptr}(\textbf{Loc}) \mid \mathit{CDptr}(\textbf{Loc}) \mid \mathit{Stoptr}(\textbf{Loc}) \\ \textbf{Stack} &\stackrel{\text{def}}{=} \textbf{Store}(\textbf{Stackvalue}) \end{aligned}$$

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

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

The internal organization of the three stores differ fundamentally. While memory and calldata use pointer structures to organize the values of reference types, storage values are accessed directly by computing the corresponding location. Thus, we model memory and calldata as stores which can contain two different types of values:

$$\begin{aligned} \textbf{Memoryvalue} &::= \mathit{Value}(\textbf{Valuetype}) \mid \mathit{Pointer}(\textbf{Loc}) \\ \textbf{Memory} &\stackrel{\text{def}}{=} \textbf{Store}(\textbf{Memoryvalue}) \\ \textbf{Calldata} &\stackrel{\text{def}}{=} \textbf{Memory} \end{aligned}$$

Storage, on the other hand is modelled as a simple mapping from locations to value types:

$$\textbf{Storage} \stackrel{\text{def}}{=} \textbf{Loc} \rightharpoonup \textbf{Valuetype}$$

Storage access is non-strict, which means that access to an undefined storage cell returns a default value. To this end, we first define a function

$$\mathit{ival} \colon \textbf{Types} \to \textbf{Valuetype}$$

which returns a default value for each value type. Then, we can define a corresponding access function for storage:

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

$$\mathit{accessStorage}(t, l, s) \stackrel{\text{def}}{=} \begin{cases} v, & \text{if } s(l) = v_\perp \\ \mathit{ival}(t), & \text{if } s(l) = \perp \end{cases}$$

Our model also provides several functions to copy structures between different stores. For example,

$$cp^m_m \colon \textbf{Loc} \times \textbf{Loc} \times \textbf{Int} \times \textbf{MTypes} \times \textbf{Memory} \times \textbf{Memory} \to \textbf{Memory}_\perp$$

$$cp^m_m(l_s, l_d, x, t, m_s, m_d) \stackrel{\text{def}}{=} \mathit{iter}'(\lambda i, m.\ \mathit{cprec}^m_m(h(l_s, \lfloor i \rfloor), h(l_d, \lfloor i \rfloor), t, m_s, m), m_d, x)$$

can be used to copy $x$ elements of type $t$ from location $l_s$ of memory/calldata $m_s$ to location $l_d$ of memory/calldata $m_d$. In the above definition $iter'$ is a function which executes a given function several times and accumulates its results.

**States** A state of a Solidity program consists of the balances of the accounts, as well as the current configuration of the different stores and the remaining amount of gas (an abstract unit of computation):

$$\mathbf{State} \stackrel{\text{def}}{=} (\mathbf{Address} \to \mathbf{Accounts}) \times (\mathbf{Address} \to \mathbf{Storage}) \times \mathbf{Memory} \times \mathbf{Stack} \times \mathbf{Nat}$$

Note that each address has its own storage. In the following, we use $acc(st)$, $sto(st)$, $mem(st)$, $sck(st)$, and $gas(st)$ to access the account, storage, memory, stack, and gas component of state $st$. Moreover, we shall use

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

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

### 2.3 Environments

Variables are always interpreted w.r.t. an environment that assigns them types and denotable-values (denvalues). They can be either a stack reference or storage reference, which denote the value of the variable, and refer to either a valuetype or a complex data type in one of the stores.

$$\begin{aligned} \mathbf{Type} &::= \mathit{Value}(\mathbf{Types}) \mid \mathit{Calldata}(\mathbf{MTypes}) \\ &\quad \mid \mathit{Memory}(\mathbf{MTypes}) \mid \mathit{Storage}(\mathbf{STypes}) \\ \mathbf{Denvalue} &::= \mathit{Stackloc}(\mathbf{Loc}) \mid \mathit{Storeloc}(\mathbf{Loc}) \end{aligned}$$

In addition to the type and value of variables, an environment contains the address of the executing contract, the address triggering the execution, and the amount of money sent with it:

$$\mathbf{Environment} \stackrel{\text{def}}{=} \mathbf{Address} \times \mathbf{Address} \times \mathbf{Valuetype} \times (\mathbf{Identifier} \rightharpoonup \mathbf{Type} \times \mathbf{Denvalue})$$

We use $address(env)$, $sender(env)$, $svalue(env)$, $denvalue(env)$ to denote the address, sender, obtained funds, and denvalue of an environment $env$.

To support the creation of fresh environments our model provides a function

$$empty : \mathbf{Address} \times \mathbf{Identifier} \times \mathbf{Address} \times \mathbf{Valuetype} \to \mathbf{Environment}$$

where $empty(a, c, s, v)$ creates a fresh environment with address $a$, contract $c$, sender $s$, value $v$, and empty type/denvalue.

To help with the declaration of new variables, our model provides a function

$$\begin{aligned}&\mathit{decl} : \mathbf{Identifier} \times \mathbf{Type} \times (\mathbf{Stackvalue} \times \mathbf{Type})_\perp \times \mathbf{Bool} \times \mathbf{Calldata} \times \mathbf{Memory}\\ &\times (\mathbf{Address} \times \mathbf{Storage}) \times \mathbf{Calldata} \times \mathbf{Memory} \times \mathbf{Stack} \times \mathbf{Environment}\\ &\rightharpoonup (\mathbf{Calldata} \times \mathbf{Memory} \times \mathbf{Stack} \times \mathbf{Environment})\end{aligned}$$

The idea is that $\mathit{decl}(id, tp, val_\perp, copy, cd, mem, sto, c, m, k, e) = (c_l, m_l, k_l, e_l)_\perp$ creates a new environment $e_l$ and corresponding calldata $c_l$, memory $m_l$, and stack $k_l$ from an existing environment $e$ with calldata $c$, memory $m$, and stack $k$. The new environment includes a new variable $id$ of type $tp$ initialized with an optional value $val$. In the case where $val$ is $\perp$, $id$ is initialized with a default value. $copy$ is a flag that indicates whether memory should be copied from the $mem$ parameter. Copying is required, for example, during external method calls. $cd$, $mem$, $sto$ are the original calldata, memory, and storage, respectively, which are used as the sources.

### 2.4 Expressions and Statements

**Expressions** Our subset of Solidity supports basic logical and arithmetic operations over signed and unsigned integers of various bit sizes. Moreover, it allows referencing elements of complex data types, creating addresses, querying the balances of addresses, or obtaining the address of the currently executing contract. Finally, it allows calling internal and external functions and obtaining the address that triggered the current execution and the value which was sent with it.

The corresponding syntax of expressions is given by a data type **E** defined as follows:

$$\begin{aligned}\mathbf{B} &::= 8 \mid 16 \mid \ldots \mid 256\\ \mathbf{L} &::= \mathit{Id}(\mathbf{Identifier}) \mid \mathit{Ref}(\mathbf{Identifier}, [\mathbf{E}])\\ \mathbf{E} &::= \mathit{SInt}(\mathbf{B}, \mathbf{Int}) \mid \mathit{UInt}(\mathbf{B}, \mathbf{Int}) \mid \mathbf{E}{+}\mathbf{E} \mid \mathbf{E}{-}\mathbf{E}\\ &\quad \mid \mathit{True} \mid \mathit{False} \mid \mathbf{E}{=}{=}\mathbf{E} \mid \mathbf{E}{<}\mathbf{E} \mid \neg\mathbf{E} \mid \mathbf{E}\wedge\mathbf{E} \mid \mathbf{E}\vee\mathbf{E}\\ &\quad \mid \mathit{Address}(\mathbf{String}) \mid \mathit{Balance}(\mathbf{E}) \mid \mathit{L}(\mathbf{L}) \mid \mathit{This} \mid \mathit{Sender} \mid \mathit{Value}\\ &\quad \mid \mathit{Call}(\mathbf{Identifier}, [\mathbf{E}]) \mid \mathit{ECall}(\mathbf{E}, \mathbf{Identifier}, [\mathbf{E}]) \mid \text{Contracts}\end{aligned}$$

where **String** denotes the type of strings, **Identifier** is just a synonym for **String**, and $[a]$ denotes a list of elements of type $a$.

Our model also provides a formal semantics for expressions. It is given in the form of a function $\mathit{expr}(exp, ev, cd, st)$ which maps an expression, environment, calldata and state to a well defined result state. This result state can be either a normal state (N) for case where an expression terminates correctly and (E) otherwise.

**Statements** The syntax of statements is given by datatype **S** defined as follows:

$$\begin{aligned}\mathbf{S} ::= & \mathit{Skip} \mid \mathbf{L} = \mathbf{E} \mid \mathbf{S}\ ;\ \mathbf{S} \mid \mathit{Ite}(\mathbf{E}, \mathbf{S}, \mathbf{S}) \mid \mathit{While}(\mathbf{E}, \mathbf{S}) \mid \mathit{Transfer}(\mathbf{E}, \mathbf{E}) \\ & \mid \mathit{Block}((\mathbf{Identifier} \times \mathbf{Type} \times E_{\perp}), \mathbf{S}) \mid \mathrm{New}(\mathbf{Identifier}, [\mathbf{E}], \mathbf{E}) \\ & \mid \mathit{Invoke}(\mathbf{Identifier}, [\mathbf{E}]) \mid \mathit{External}(\mathbf{E}, \mathbf{Identifier}, [\mathbf{E}], \mathbf{E})\end{aligned}$$

Again, our model provides a formal semantic for statements in the form of a function $stmt(sm, ev, cd, st)$, which maps a statement, environment, calldata, and state to a well defined result state.

One statement which is of particular interest for this work is assignments which alter the stores. Assignments are special in Isabelle/Solidity when complex data types, such as arrays, are involved. In particular, if the left- and right-hand sides are both located in memory, then the assignment only changes the pointer. However, if one of the two is storage and the other is memory, then the assignment executes an actual copy.

## 3 Defining Type Safety

We define type safety of Solidity over contract environments and their respective stores. A given environment $ev$ is considered type safe with respect to a set of accounts $acc$, a stack $sck$, a memory $mem$, a storage $sto$ and a calldata $cd$ if they satisfy the following function:

$$\begin{aligned}\mathit{TypeSafe}(ev, acc, sck, mem, sto, cd) \stackrel{\text{def}}{=}\ & \mathit{TypeConsistency}(ev, sck, mem, cd, sto) \\ & \wedge \mathit{UniqStLocs}(\mathit{denvalue}(ev)) \ \wedge \mathit{CompPnts}(sck, \mathit{sto}(\mathit{address}(ev)), \mathit{denvalue}(ev)) \\ & \wedge \mathit{CompPnts}(sck, mem, \mathit{denvalue}(ev)) \wedge \mathit{CompPnts}(sck, cd, \mathit{denvalue}(ev)) \\ & \wedge \mathit{LessTLocs}(sck) \wedge \mathit{LessTLocs}(mem) \wedge \mathit{LessTLocs}(cd) \\ & \wedge \mathit{SafeCont}(sto, \mathbf{Env_P}) \ \wedge \mathit{MNoPrefxs}(\mathbf{Env_P}) \ \wedge \mathit{VBalT}(acc) \\ & \wedge \mathit{VSTypes}(\mathit{svalue}(ev))\end{aligned}$$

### 3.1 Type Consistency

Type consistency is the key property we consider when defining type safety and its definition requires considering two key components. The stores contained within the state, which contain the values of variables (Sec. 2.2), and the *denvalue* of the environment, which is its type environment and maps their variables to their store locations and types (Sec. 2.3).

Type consistency of an environment and a state's stores requires that the types of the variables specified in *denvalue* are consistent with their actual values in the stores. Recall that the **Valuetype**, which represent the values which are stored, is synonymous with the **String** type (Sec. 2.1). As a result of this our storage model is very generalized with all the contents being of **String** type. Thus, in

our context, type consistency can only guarantee that the content of a given store location (associated with a variable in the *denvalue*) can be correctly interpreted to a value of its type. Notably this means that locations which have become disconnected from the *denvalue*, floating pointers, are not checked. However, this is not a problem for our semantics as these values should not be accessible using the statements and expressions we currently

**Type consistency of value types** To determine whether a given string value conforms to a given type we define the function

$$\mathit{TypeCon}\colon \mathbf{Types} \times \mathbf{Valuetype} \to \mathbf{Bool}, \text{ where}$$

$$\mathit{TypeCon}(t, v) \stackrel{\text{def}}{=} \begin{cases} (v = \text{“}\mathit{True}\text{”} \vee v = \text{“}\mathit{False}\text{”}) & \text{if } t = \mathit{TBool} \\ \text{“.”} \notin v & \text{if } t = \mathit{TAddr} \\ \boxed{1} & \text{if } t = \mathit{TUInt}(b) \\ \boxed{2} & \text{if } t = \mathit{TSInt}(b) \end{cases}$$

$$\boxed{1} = \lceil v \rceil \geq 0 \wedge \lceil v \rceil < 2^b \wedge \lfloor \lceil v \rceil \rfloor = v$$

$$\boxed{2} = \lceil v \rceil \geq -(2^{b-1}) \wedge \lceil v \rceil < 2^{b-1} \wedge \lfloor \lceil v \rceil \rfloor = v$$

The cases where type $t$ is a boolean or address are trivial, in the case of booleans we require the string $v$ to be either “*True*” or “*False*” and for addresses we just require that the address does not contain a “." to prevent conflicts with our memory address indexing method. For unsigned integers, the integer representation of $v$ must be greater than or equal to 0 and must also be less than the maximum value that can be represented by the bit size $b$. Similarly, unsigned integers must fall between the minimum and maximum values which can be supported by size $b$. In addition to the size constraint there is an additional property that string representations of integers must hold. Converting $v$ to an integer and then back to a string must remain the same. The reason for this is that Solidity does not allow numeric values with preceding zeros, but this restriction is not present for our stores which only operate over **Valuetype**. It is therefore possible to have the string "0001" in a store. This property of TypeCon prevents this for type consistent stores.

**Type consistency of reference types** In addition to the value types we also support *reference types*. These types include memory, calldata and storage arrays and also storage maps. The concept of verifying these types is much the same, involving traversing the structure of the reference type to check every **Valuetype** element it contains. The major difference is with the traversal mechanism as memory and calldata support pointers while storage accesses locations directly.

Note that we only consider a *reference type* structure to be type consistent if all of its elements are type consistent. We do not allow for partial conformity. We also do not consider partially initialized arrays to be type safe, this is because Solidity initializes all arrays with default values on declaration and thus every location of an array must have a value.

The function *MCon* is defined to check the type consistency for memory and calldata arrays for a type $t$, memory (or calldata) $m$ and location $loc$.

$$MCon\colon \mathbf{MTypes} \times \mathbf{Memory} \times \mathbf{Loc} \to \mathbf{Bool},\ \text{where}$$

$$MCon(t, m, loc) \stackrel{\text{def}}{=} \begin{cases} \boxed{1} & \text{if } t = MTValue(typ) \\ \boxed{2} & \text{if } t = MTArray(len, subTyp) \end{cases}$$

$$\boxed{1} = \begin{cases} TypeCon(t,\ v) & \text{if } accessStore(loc,\ m) = Value(v) \\ False & \text{otherwise} \end{cases}$$

$$\boxed{2} = \forall i < len \begin{cases} subTyp = MTValue(t') & \text{if } accessStore(h(loc, i),\ m) \\ \wedge TypeCon(t',\ v) & = Value(v) \\ subTyp = MTArray(len', subTyp') & \text{if } accessStore(h(loc, i),\ m) \\ \wedge MCon(subTyp', m, ptrLoc) & = Pointer(ptrLoc) \\ False & \text{otherwise} \end{cases}$$

*MCon* first distinguishes between whether the type being examined is an *MTArray* or an *MTValue*. In the *MTValue* case, the string value $v$ located at $loc$ in $m$ is retrieved. If this is a **Valuetype**, then the $v$ is checked against type t using *TypeCon*. If the accessed value is not a **Valuetype**, for example a *Pointer*, *MCon* returns false. The reason for this is that, while a *Pointer* is a **String** in the store, it is not considered a valid value, instead indicating the address prefix of the next array. For the *MTArray* case the $loc$ which is passed to *MCon* represents the prefix location for all elements in the array to be checked. We access each element individually using a location hash (Eq. 1), where the index $i$ is limited to be less than the length of the array $len$, due to the zero indexing of arrays. If the accessed value $v$ is a **Valuetype**, then the subtype of the array, must be an *MTValue* and *TypeCon* must hold. Otherwise, if the accessed value is a *Pointer* to the start of the next sub array, the subType must be an *MTArray* type and each index of this sub array must satisfy *MCon*. Finally, if accessing the index returns nothing, then *MCon* returns false. This prohibits both null pointers and also partially initialized arrays.

To verify storage locations, a similar function *SCon* is defined:

$$SCon\colon \mathbf{STypes} \times \mathbf{Loc} \times \mathbf{Storage} \to \mathbf{Bool}, \text{where}$$

$$SCon(t, loc, stor) \stackrel{\text{def}}{=} \begin{cases} \boxed{1} & \text{if } t = (STValue\ typ) \\ \boxed{2} & \text{if } t = (STArray\ len\ subTyp) \\ \boxed{3} & \text{if } t = (STMap\ keyT\ subTyp) \end{cases}$$

$$\boxed{1} = TypeCon(typ, accessStorage(typ, loc, stor))$$

$$\boxed{2} = \forall i < len.SCon(subTyp, (h\ loc\ i), stor)$$

$$\boxed{3} = \forall i :: string.(typeCon\ keyT\ i) \implies SCon(subTyp, (hloci), stor)$$

For the array case, *SCon* operates similarly to *MCon*, however, as storage does not use pointers to reference structures there is no need to look up a pointer

before indexing the array. The case for *STValues* is also similar to *MCon*, however, there is no need to check the return of *accessStorage* as it will always return either an existing value or the default value of the type *typ* which is type consistent by construction. Mappings (*STMap*) are unique to storage and support mappings from a key to a value. The key can be any value but must conform to the key type (*keyT*). Additionally, a key must always have a value. Thus, *STMaps* are considered type consistent if *TSSCon* holds for all key values $i$, which conform to *keyT*.

**Type consistency of environments and stores** Finally, the following function determines whether the values in a set of stores are type consistent with an environment.

$$\begin{aligned}
&\mathit{TypeConsistency}\colon \mathbf{Environment} \times \mathbf{Stackvalue} \times \\
&\quad \mathbf{Memory} \times \mathbf{Calldata} \times \mathbf{Storage} \to \mathbf{Bool}, \text{ where} \\
&\mathit{TypeConsistency}(ev, sck, mem, cd, stor) \stackrel{\text{def}}{=} \\
&\forall (t, l) \in \mathit{range}(\mathit{denvalue}(ev)) \implies \\
&\begin{cases} \boxed{1} & \text{if } l = \mathit{Stackloc}(loc) \\ t = \mathit{Storage}(typ) \wedge \mathit{SCon}(typ, ptr, \mathit{stor}(\mathit{address}(ev))) & \text{if } l = \mathit{Storeloc}(loc) \end{cases}
\end{aligned}$$

Type consistency checks the types $t$ and locations $l$ for all the variables stored within the range of the *denvalue* of the environment $ev$. We then distinguish between variables which are referenced from the stack and those stored in storage. For locations on the stack we do the following:

$$\boxed{1} = \begin{cases}
t = \mathit{Value}(typ) \wedge \mathit{TypeCon}(loc, typ) & \\
\qquad\qquad \text{if } \mathit{accessStore}(loc, sck) = \mathit{Simple}(val) & \\
t = \mathit{Memory}(typ) \wedge \mathit{MCon}(typ, mem, ptr) & \\
\qquad\qquad \text{if } \mathit{accessStore}(loc, sck) = \mathit{Memptr}(ptr) & \\
t = \mathit{Calldata}(typ) \wedge \mathit{MCon}(typ, cd, ptr) & \\
\qquad\qquad \text{if } \mathit{accessStore}(loc, sck) = \mathit{CDptr}(ptr) & \\
t = \mathit{Storage}(typ) \wedge \mathit{SCon}(typ, ptr, \mathit{stor}(\mathit{address}(ev))) & \\
\qquad\qquad \text{if } \mathit{accessStore}(loc, sck) = \mathit{Stoptr}(ptr) &
\end{cases}$$

First, we look up the corresponding value from the stack using $l$ and then further distinguish between four cases. In the case where a *Simple* element is stored, the corresponding type $t$ must be a *Value* type. The string value *val* is then checked against the type *typ* using *TypeCon*. Alternatively, the $l$ could correspond to a pointer *ptr* for a *reference type*. In these cases, $t$ must be one of the store types and correspond to the same store as *ptr*. Moreover, the structure of the *ptr* in the store must correspond to the type which is checked using the appropriate function. Note that in the case of storage pointer, the storage for the current contract, which is using $ev$, is used $\mathit{stor}(\mathit{address}(ev))$. The process is similar for

cases where $l$ is a storage location. The only difference is that we do not need to reference a pointer before checking the structure with *SCon* as the location points to storage directly.

### 3.2 Structural Consistency

In addition to ensuring that the stores are consistent with each entry in the type environment, it is also important to ensure that the type environment is consistent with itself and the layout of the store. To ensure this, we have a number of properties to ensure that structual consistency is maintained.

This includes ensuring that if two variables in the denvalue point to the same stack location, then their types must be the same (*UniqStLocs*). Further, there must be consistency between the stack and the stores (*CompPnts*). If two variables with different *Stacklocs* contain pointers to the same store and the pointers have the same location or are sublocations of one another then their types must be the same or compatible. This is true for pointers to all three stores. We also enforce that memory, calldata and stacks must not contain any values at locations which are greater than the top location of the store (*LessTLocs*).

### 3.3 Contract Related Properties

Smart contracts can have member variables which are always stored in storage (using **STypes**). When a contract is initialized these member variables are loaded into the store and their references are added to the type environment. Therefore, we require that all contracts in the contract environment $\mathbf{Env_P}$ must have type consistent member variables (*SafeCont*) with respect to their stores. Additionally, member variables should not reference each other, as when they are defined in Solidity (before the contract is created), this would not be possible (*MNoPrefxs*). Contracts also have a balance which stores the current amount of cryptocurrency associated with a contract and a *evalue* which represents the funds sent to the contract at the time of a contract call. We require that both of these values must conform to an appropriate type (*VBalT*, *VSTypes*).

## 4 Verifying Type Safety

Type safety of Solidity can primarily be violated in one of two ways: A statement or expression changes the values in the stores, i.e. assignments. A statement or expression alters the denvalue, i.e. a new variable is declared or a new environment is created.

Statements are the primary way in which these alterations may occur and so we must verify that the type safety of the environment is preserved after each statement. Resulting in the following lemma:

**Lemma 1 (TypeSafe_Statements).**

$$TypeSafe(ev, acc(st), sck(st), mem(st), sto(st), cd) \wedge \tag{2}$$
$$stmt(smt, ev, cd, st) = \mathrm{N}((), st') \tag{3}$$
$$\implies TypeSafe(ev, acc(st'), sck(st'), mem(st'), sto(st'), cd) \tag{4}$$

Given an environment $ev$ which is type safe with respect to the accounts, stack, memory and storage of a state $st$ and of calldata $cd$ (Eq. 2). Then, for every statement which terminates normally and returns an updated state $st'$ (Eq. 3), $ev$ remains type safe with respect to the stores in the new $st'$ and the original calldata $cd$, which remains un-changed (Eq. 4).

*Proof.* The proof is by induction over the statements with arbitrary $st'$. For non-trivial cases each statement is then proven by constructing a series of abstract states which follow the definition of the current statement while demonstrating that each state preserves type safety. □

Statements rely on expressions when working with variables in the state. Thus, we also proved the following lemma:

**Lemma 2 (TypeSafe_Expressions).**

$$TypeSafe(ev, acc(st), sck(st), mem(st), sto(st), cd) \wedge \tag{5}$$
$$expr(exp, ev, cd, st, g) = \mathrm{N}((v, t), g') \implies \tag{6}$$
$$\begin{cases} TypeCon(loc, typ) \wedge v = Simple & \text{if } t = Value(typ) \\ MCon(typ, mem(st), ptr) \wedge v = Memptr & \text{if } t = Memory(typ) \\ MCon(typ, cd, ptr) \wedge v = CDptr & \text{if } t = Calldata(typ) \\ SCon(typ, ptr, sto(st)(address(ev))) \wedge v = Stoptr & \text{if } t = Storage(typ) \end{cases} \tag{7}$$

Given a type safe environment $ev$ (Eq. 5), the lemma states that for an expression $exp$ which terminates normally and returns a value $v$ and type $t$ (Eq. 6), $v$ is indeed a compatible string representation for a variable of type $t$ (Eq. 7).

*Proof.* Similar to TypeSafe_Statements, this lemma is proven by induction over the expressions and shown for each case. □

In the following we discuss some of the more complex aspects of the proof.

### 4.1 Verifying Memory/Calldata Reference Types

If a location containing a simple value type is altered, the only requirement is to show that the new value is consistent with the type of the location in the denvalue. However, for arrays the requirements are more complex. The reason for this is that not only is it necessary to verify the location that has changed is consistent, it is also necessary to check that any variables that may reference the location also remain consistent.

One particularly interesting case is when assigning a memory array with a calldata array as a value. During this process the array is copied from calldata to memory. This is handled by the function $cp^m_m$ discussed in Sec. 2.2. Thus, we needed to verify that $cp^m_m$ does not violate type safety (We verified similar properties for all other copy functions).

To demonstate that executing $cp^m_m$ does not violate type safety it is only necessary to show that the segment of the destination memory which is altered by $cp^m_m$ is *MCon*. Essentially, this requires showing that the result of $cp^m_m$ is *MCon* for the structure that has been copied at the location in the destination it was copied to. To this end, we verified the following lemma.

**Lemma 3 (MCon_cpm2m).**

$$MCon(MTArray(x,t), ms, ls) \ \wedge \ x > 0 \ \wedge \tag{8}$$
$$cp^m_m(ls, ld, x, t, ms, md) = updM \ \wedge \tag{9}$$
$$\implies MCon(MTArray(x,t)aa, updM, ld) \tag{10}$$

Eq. 8 establishes that the source memory/calldata store $ms$ is *MCon* with respect to an *MTArray* type of length $x$ and subtype $t$ at the prefix location $ls$. Further, $x$ is a nat greater than zero, as array lengths must be greater than zero. Eq. 9 then states that $cp^m_m$, which copies the sub-elements of the $MTArray(x,t)$ from $ms$ to the destination memory (or calldata) $md$ to location $ld$, terminates normally and returns an updated memory/calldata $updM$. Finally, Eq. 10 shows that $updM$ is *MCon* with respect to the $MTArray(x,t)$, but at the destination location $ld$.

*Proof.* As $cp^m_m$ is a mutually recursive definition using $iter'$ and $cprec^m_m$ we first expand the definition of $cp^m_m$. We then perform induction over the length of the $x$ which are all the indices of the source array being copied. For the non-trivial case ($x > 0$) we then apply structural induction over the **MTypes** ($t$). For cases where $t$ is a further *MTArray* we again apply an additional induction over the sub-arrays length for $iter'$. The intuition here is that if $cprec^m_m$ correctly reconstructs a copy of the source structure, and that structure was *MCon* in the source, then the structure should also be *MCon* in the destination. □

### 4.2 Internal and External Method Calls

Internal and External method calls (*ECall*, *Call*) are another interesting aspect of the verification. When calling external methods, a new state is created with an empty stack and memory, and a new environment with a new denvalue in which the external contracts member variables are loaded. This state and environment are then used to load the method parameters, using *load* and *decl*.

The result of this loading process is then used to execute the method body. As method bodies are defined using expressions, Lemma 2 can be used to show that their return values are type consistent. However, in order to use Lemma 2, we must show that the environment and stores being used for the execution of the method body are type safe.

To verify that the new environment is type safe we proved three lemmas: $ffoldInitTypeSafe$ which confirms that the fresh environment is type safe, and *TypeSafeDecl* and *TypeSafeLoad* which verify that the environment remains type safe after the variables have been loaded. The lemma $ffoldInitTypeSafe$ is trivial as an environment with only contract member variables is by definition type safe, and so we will focus on *TypeSafeDecl* and *TypeSafeLoad*.

**Lemma 4 (TypeSafeLoad).**

$$TypeSafe(lev0, acc(lst), sck(lst), mem(lst), sto(lst), lcd0) \wedge \tag{11}$$
$$TypeSafe(lev, acc(lst), lk, lm, sto(lst), lcd) \wedge \tag{12}$$
$$\forall locs\ typs.\ \neg\ lcp \implies MCon(tp, mem(lst), locs) \implies MCon(tp, lm, locs) \tag{13}$$
$$(\forall ev\ cd\ k\ m\ g'.\ load(lcp, lis, lxs, lev0, lcd0, lk, lm, lev, lcd, lst, lg) = \mathrm{N}((ev, cd, k, m), g') \implies TypeSafe(ev, acc(lst), k, m, sto(lst), cd)) \tag{14}$$

Eq. 11 and Eq. 12 state that the source and destination environments are type safe with respect to the accounts, stack, memory and storage of a state *lst* and of calldata *lcd0* and *lcd* respectively. Eq. 13 states that for internal method calls, where a copy is not taking place ($\neg$ *lcp*), all locations and types which are *MCon* for the source memory are also *MCon* for the destination memory. Finally, equation Eq. 14 states that for results of *load* which terminate in a N state, *load* returns an environment *ev* which is *TypeSafe* with respect to the returned calldata *cd*, stack $k$, memory $m$ and the storage and accounts of *lst*.

*Proof.* The proof is by induction over the elements of the list of variables to be loaded (*lis*). The base case (empty list) is trivial as the destination elements *lev lcd lm lk* are returned. For the inductive case it is necessary to show that after the current head of the list is loaded using *decl* the resultant environment is type safe, this is verified using Lemma 5. When using the *TypeSafeDecl* lemma we pass the assumptions of *TypeSafeLoad*. In addition to the knowledge that the current value and type $(v, t'')$ being declared (the head of list *lxs*) is the result of an expression and so is typeCon using Lemma 2. □

**Lemma 5 (TypeSafeDecl).**

$$TypeSafeLoad(amms) \wedge \tag{15}$$
$$\forall ts.t_p \neq \mathbf{Storage}(ts) \wedge \tag{16}$$
$$decl(i_p, t_p, (v, t''), lcp, lcd, mem(lst), (sto(lst)address(lev0)), (lcd0, lm, lk, lev0)) = \mathrm{N}((e, c', k', m'), g') \tag{17}$$
$$\implies TypeSafe(e, acc(lst), k', m', sto(lst), c') \wedge \tag{18}$$
$$(\forall locs\ typs.\ \neg\ lcp \implies MCon(tp, mem(lst), locs) \implies MCon(tp, m', locs)) \tag{19}$$

As decl is called from the context of load, we pass the assumptions of *TypeSafeLoad* to *TypeSafeDecl* (Eq. 15). In addition, Eq. 16 establishes that the type being

added to the denvalue $t_p$ is not a **Storage** type. Then, Eq. 17 requires that *decl* terminates in a N state and returns an environment $e$ calldata $c'$, stack $k'$ and memory $k'$. Eq. 18 concludes that $e$ is *TypeSafe* with respect to $k'$, $c'$, $m'$, and the accounts and storage of the source state *lst*. Finally, Eq. 19 states that for internal method calls all locations and types which are *MCon* in the source memory $mem(lst)$ are also *MCon* in the resultant memory $m'$.

*Proof.* The proof for *TypeSafeDecl* is a case split over the outcome of *decl*. Trivial cases such as when $t_p$ is a **Valuetype** are proven by unfolding the definitions and more complex cases, such as declaring array types are handled using the MCon_cp(m/s)2m lemmas which ensures the changes being made are MCon. □

## 5 Related Work

*Verification of type safety in Isabelle* Type safety has been formalized and verified in Isabelle. One famous example is the formalization and verification of the soundness of a static type system for IMP [25]. In addition, there have been verification of type safety aspects for real programming languages in Isabelle, such as Java [27], C++ [28], ecc. Compared to traditional programming languages, Solidity provides some specialized features, such as the different types of stores. Thus, by providing a formalization of type safety for Solidity we complement this line of research.

*Formalizations of Solidity* Another line of research which is related to our work concerns formalizations of Solidity. As outlined by Almakhour et al. [3] and Tolmach et al. [26], there is a growing amount of research investigating the formalization of Solidity. Early work in this area was done by Bhargavan et al. [9] who describe an approach to map a Solidity contract to F* where it can then be verified. TinySol [6] and Featherweight Solidity[12], on the other hand, are two calculi formalizing some core features of Solidity. Crosara et al. [13] describe an operational semantics for a subset of Solidity. Moreover, Ahrendt and Bubel describe SolidiKeY [2], a formalization of a subset of Solidity in the KeY tool [1] to verify data integrity for smart contracts. In addition, Jiao et al. [15,16], provide a formalization of Solidity in $\mathbb{K}$. While all of these works focus on the formalization of Solidity, none of them investigate type safety aspects.

*Verification of type safety for Solidity* Crafa et al. [12] investigate soundness of a static type system for Featherweight Solidity (a formalization of a subset of Soldity). In their work, they identify problems with the Solidity type system and propose an alternative one. Our work differs in two main aspects from their work. First, they focus on the verification of soundness of the base types of Solidity, with our work we also focus on verifying consistency of the complex types, such as, memory arrays and their pointer structures. Second, FS is a restricted subset of Solidity which lacks many features of modern Solidity. For example FS does not support the various types of stores which are available to a Solidity program and which pose a particular challenge to the verification of type safety.

## 6 Conclusion

With this paper we describe our work on a type-safe version of Solidity. To this end, we first provide a formalization of type-safety for Solidity programs. Then we verify that our semantics of Solidity preserves type-safety in Isabelle.

*Technical Challenges* One of the key technical challenges in verifying type safety for Solidity is the complexity of the different stores and the pointer/addressing scheme used for reference types in Solidity. This is even more pronounced when verifying the type safety of statements such as Assign, which have many cases and a semantics which changes dependent on the storage types involved. We found that the interactions between the different stores made deriving properties for type safe environments very difficult and that verification of those properties required the largest amount of the proof effort.

We have covered some of the more complex aspects in this paper. The formalization of the *MCon* and *SCon* properties which are able to check the type consistency of memory and storage reference types. We also examined the verification effort for ensuring the type consistency of copying between the different stores (Lemma 3) and declaring and loading new states and environments which contain reference types (Lemma 4, Lemma 5).

*Type issues in Isabelle/Solidity* While we did not identify any type issues of Solidity as a language for our definition of type safety however we did detect a number of issues in our formalization. In total, we found 13 issues with our formalization, 12 of these were related to missing type checks without which the type safety of the environment could be violated. The remaining issue was a bug in the operation of the semantics. Importantly 10 of the 13 issues were related to functions which operated over reference types, demonstrating the complexity of these types. To highlight some of these issues:

- $cprec^m_m$ did not correctly traverse the pointer structure of the source memory/calldata. While we accessed the pointers from the store we did not use those pointers as the prefix for the next indexed location. As a result any pointer which did not point to itself would not have been reached.
- *Call* was able to accept storage reference types as parameters. This is prohibited in Solidity.
- *decl* did not verify that the type of the variable being declared in storage matched the type of the value being added to the *denvalue*. This would result in mismatched types between the stores and the *denvalue*.

*Future work* The type system discussed in this paper uses untyped stores in combination with a typing environment and can only be checked at runtime. Thus, future work should focus on the development of a static, strongly typed, type system which can also be checked at compile time.

*Acknowledgments* We thank the anonymous reviewers of ICTAC 2024 for their careful reading and constructive comments to improve an earlier version of this paper. This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1].

## References


1. Ahrendt, W., Beckert, B., Bubel, R., Hähnle, R., Schmitt, P.H., Ulbrich, M.: Deductive software verification–the KeY book, vol. 10001. Springer (2016). https://doi.org/10.1007/978-3-319-49812-6
2. Ahrendt, W., Bubel, R.: Functional verification of smart contracts via strong data integrity. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation: Applications. pp. 9–24. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-61467-6_2
3. Almakhour, M., Sliman, L., Samhat, A.E., Mellouk, A.: Verification of smart contracts: A survey. Pervasive and Mobile Computing **67**, 101227 (2020). https://doi.org/10.1016/j.pmcj.2020.101227
4. Authors, S.: Solidity developer survey 2023 results (2024), https://soliditylang.org/blog/2024/04/03/solidity-developer-survey-2023-results/
5. Azaria, A., Ekblaw, A., Vieira, T., Lippman, A.: Medrec: Using blockchain for medical data access and permission management. In: 2016 2nd International Conference on Open and Big Data (OBD). pp. 25–30 (2016). https://doi.org/10.1109/OBD.2016.11
6. Bartoletti, M., Galletta, L., Murgia, M.: A minimal core calculus for Solidity contracts. In: Pérez-Solà, C., Navarro-Arribas, G., Biryukov, A., Garcia-Alfaro, J. (eds.) Data Privacy Management, Cryptocurrencies and Blockchain Technology. pp. 233–243. Springer (2019). https://doi.org/10.1007/978-3-030-31500-9_15
7. Berghofer, S., Wenzel, M.: Inductive datatypes in hol — lessons learned in formal-logic engineering. In: Bertot, Y., Dowek, G., Théry, L., Hirschowitz, A., Paulin, C. (eds.) TPHOLs. pp. 19–36. Springer (1999). https://doi.org/10.1007/3-540-48256-3_3
8. Bertoni, G., Daemen, J., Peeters, M., Van Assche, G.: Keccak. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT. pp. 313–314. Springer (2013). https://doi.org/10.1007/978-3-642-38348-9_19
9. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. In: Programming Languages and Analysis for Security. p. 91–96. PLAS, ACM (2016). https://doi.org/10.1145/2993600.2993611
10. Chavez-Dreyfuss, G.: Sweden tests blockchain technology for land registry. https://www.reuters.com/article/us-sweden-blockchain-idUSKCN0Z22KV, accessed: 2023-04-18
11. Clegg, P., Jevans, D.: Cryptocurrency crime and anti-money laundering report. Tech. rep., CipherTrace (2021)
12. Crafa, S., Di Pirro, M., Zucca, E.: Is Solidity solid enough? In: Bracciali, A., Clark, J., Pintore, F., Rønne, P.B., Sala, M. (eds.) Financial Cryptography and Data Security. pp. 138–153. Springer (2020). https://doi.org/10.1007/978-3-030-43725-1_11
13. Crosara, M., Centurino, G., Arceri, V.: Towards an Operational Semantics for Solidity. In: van Rooyen, J., Buro, S., Campion, M., Pasqua, M. (eds.) VALID. pp. 1–6. IARIA (Nov 2019)
14. Ethereum: Solidity. https://docs.soliditylang.org/, accessed: 2023-05-04
15. Jiao, J., Kan, S., Lin, S.W., Sanan, D., Liu, Y., Sun, J.: Semantic understanding of smart contracts: executable operational semantics of Solidity. In: SP. pp. 1695–1712. IEEE (2020). https://doi.org/10.1109/SP40000.2020.00066

16. Jiao, J., Lin, S.W., Sun, J.: A generalized formal semantic framework for smart contracts. In: Wehrheim, H., Cabot, J. (eds.) FASE. pp. 75–96. Springer (2020). https://doi.org/10.1007/978-3-030-45234-6_4
17. Kelly, J.: Banks adopting blockchain 'dramatically faster' than expected: IBM. https://www.reuters.com/article/us-tech-blockchain-ibm-idUSKCN11Y28D (2016), accessed: 2023-05-04
18. Llama, D.: Tvl breakdown by smart contract language (jun 2024), https://defillama.com/languages
19. Marmsoler, D., Brucker, A.D.: A Denotational Semantics Of Solidity In Isabelle/HOL. In: Software Engineering and Formal Methods: 19th International Conference, SEFM 2021, Virtual Event, December 6–10, 2021, Proceedings. pp. 403–422. Springer-Verlag, Berlin, Heidelberg (2021). https://doi.org/10.1007/978-3-030-92124-8_23
20. Marmsoler, D., Brucker, A.D.: Conformance Testing of Formal Semantics Using Grammar-Based Fuzzing. In: Kovács, L., Meinke, K. (eds.) Tests and Proofs. pp. 106–125. Springer International Publishing, Cham (2022). https://doi.org/10.1007/978-3-031-09827-7_7
21. Marmsoler, D., Brucker, A.D.: Isabelle/solidity: A deep embedding of solidity in isabelle/hol. Archive of Formal Proofs (July 2022), https://isa-afp.org/entries/Solidity.html, Formal proof development
22. Mendling, J., Weber, I., Aalst, W.V.D., Brocke, J.V., Cabanillas, C., Daniel, F., Debois, S., Ciccio, C.D., Dumas, M., Dustdar, S., Gal, A., García-Bañuelos, L., Governatori, G., Hull, R., Rosa, M.L., Leopold, H., Leymann, F., Recker, J., Reichert, M., Reijers, H.A., Rinderle-Ma, S., Solti, A., Rosemann, M., Schulte, S., Singh, M.P., Slaats, T., Staples, M., Weber, B., Weidlich, M., Weske, M., Xu, X., Zhu, L.: Blockchains for business process management - challenges and opportunities. ACM Trans. Manage. Inf. Syst. **9**(1) (feb 2018). https://doi.org/10.1145/3183367
23. Nakamoto, S.: Bitcoin: A Peer-to-Peer Electronic Cash System (2008). https://doi.org/10.2139/ssrn.3440802
24. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: A Proof Assistant for Higher-Order Logic (2002). https://doi.org/10.1007/3-540-45949-9
25. Nipkow, T., Klein, G.: Concrete semantics: with Isabelle/HOL. Springer (2014). https://doi.org/10.1007/978-3-319-10542-0
26. Tolmach, P., Li, Y., Lin, S.W., Liu, Y., Li, Z.: A survey of smart contract formal specification and verification. ACM Comput. Surv. **54**(7) (jul 2021). https://doi.org/10.1145/3464421
27. Von Oheimb, D., Nipkow, T.: Machine-checking the java specification: Proving type-safety. In: Formal Syntax and Semantics of Java, pp. 119–156. Springer (2002). https://doi.org/10.1007/3-540-48737-9_4
28. Wasserrab, D., Nipkow, T., Snelting, G., Tip, F.: An operational semantics and type safety proof for multiple inheritance in c++. In: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications. pp. 345–362 (2006). https://doi.org/10.1145/1167515.1167503
29. YCharts.com: Ethereum transactions per day. https://ycharts.com/indicators/ethereum_transactions_per_day, accessed: 2024-05-04
30. Yurcan, B.: How blockchain fits into the future of digital identity. https://fintechranking.com/2016/04/10/how-blockchain-fits-into-the-future-of-digital-identity/ (2016)