Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL

DIEGO MARMSOLER and ACHIM D. BRUCKER, University of Exeter, UK

Smart contracts are computer programs designed to automate legal agreements. They are usually developed in a high-level programming language, the most popular of which is Solidity. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions. As for 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.

This paper addresses the issue of smart contract vulnerabilities by introducing an executable denotational semantics for Solidity within the Isabelle/HOL interactive theorem prover. This formal semantics serves as the basis for an interactive program verification environment for Solidity smart contracts. To evaluate our semantics, we integrate grammar-based fuzzing with symbolic execution to automatically test it against the Solidity reference implementation. The paper concludes by showcasing the formal verification of Solidity programs, exemplified through the verification of a basic Solidity token.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Program semantics**; • **Security and privacy → Logic and verification**.

Additional Key Words and Phrases: Solidity, Denotational Semantics, Isabelle/HOL

ACM Reference Format:
Diego Marmsoler and Achim D. Brucker. 2018. Isabelle/Solidity: A deep embedding of Solidity in Isabelle/HOL. 1, 1 (November 2018), 56 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Blockchain [42] is a novel technology to store data in a decentralised way without the need of a trusted third party. Although the technology was originally invented to enable cryptocurrencies, it quickly found applications in several other domains, such as *finance* [33], *healthcare* [21], *land management* [15], *business process management* [23], and even *identity management* [57]. According to McKinsey, blockchain had a market capitalisation of more than $150B in 2018 [22] and Gartner predicts a business value of the technology of $3.1T by 2030 [24].

One important innovation which comes with blockchain are *smart contracts*. These are digital contracts which are automatically executed once certain conditions are met and which are used to automate transactions on the blockchain. For instance, a payment for an item might be released instantly once the buyer and seller have met all specified parameters for a deal. Every day, hundreds of thousands of new contracts are deployed managing millions of dollars' worth of transactions [56].

Technically, a smart contract is code which is deployed to a blockchain and which can be executed by sending special transactions to it. Thus, as for every computer program, smart contracts may contain bugs which can be exploited. However, since smart contracts are often used to automate

Authors' address: Diego Marmsoler, d.marmsoler@exeter.ac.uk; Achim D. Brucker, a.brucker@exeter.ac.uk, University of Exeter, University Of Exeter Campus, Exeter, Devon, UK, EX4 4RN.

ACM XXXX-XXXX/2018/11-ART
https://doi.org/XXXXXXX.XXXXXXX

financial transactions, such exploits may result in huge economic losses. For example, in 2016 a vulnerability in an Ethereum smart contract was exploited resulting in a loss of approximately $60M [7]. More recently, hackers exploited a vulnerability in the DeFi-platform Poly Network to steal $600M [43]. As another example, an incorrectly initialised contract was the root cause of the Parity Wallet bug that froze $280M [46]. In general, it is estimated that since 2019, more than $5B was stolen due to vulnerabilities in smart contracts [16].

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. Thus, companies have specialised to provide dedicated auditing services in which experts analyse smart contracts for vulnerabilities. Unfortunately, however, even rigorous security audits cannot guarantee correctness of smart contracts as shown in a more recent attack in which $31M was stolen from a smart contract which had received already three security audits throughout the year [25].

Smart contracts are usually developed in a high-level programming language, the most popular of which is *Solidity* [5]. Solidity has been designed for use with the Ethereum Virtual Machine (EVM) and thus it works on all EVM-based smart contract platforms, such as Ethereum, Avalanche, Moonbeam, Polygon, BSC. As of today, more than 90% of all smart contracts are developed using Solidity [36] and according to a 2023 survey, Solidity is by far the most popular language used by blockchain developers (in fact it is twice as popular as second most popular language) [6].

To provide a first impression of the language, we briefly discuss a simple smart contract in Solidity which allows clients to deposit and withdraw funds:

```
contract Bank {
  mapping(address => uint256) balances;

  function deposit() public payable {
    balances[msg.sender] = balances[msg.sender] + msg.value;
  }

  function withdraw() public {
    msg.sender.transfer(balances[msg.sender]);
    balances[msg.sender] = 0;
  }
}
```

The example shows several specialities of the Solidity programming language. In particular, every contract has an internal balance and funds can be transferred to and from this balance either explicitly (by transferring funds) or implicitly (via an external method call). Additionally, the type system provides, e.g., numerous integer types of different sizes (e.g., `uint256`) and the Solidity programs can make use of different types of stores for data (e.g., storage and memory).

In this paper, we address the problem of developing correct smart contracts in Solidity. To this end, we present an executable denotational semantics for Solidity in the interactive theorem prover Isabelle/HOL [45]. Our approach combines an expressive logic, i.e., higher-order logic (HOL) within an interactive theorem prover with a testing framework allowing us to validate the formalization against the actual implementation. This combination enables us to quickly analyse the impact of changes to the semantics while ensuring formal consistency and compliance to the implementation. The ability to quickly assess changes in Solidity is important, as Solidity is a fast evolving language.

The Solidity manual [5], e.g., states: “When deploying contracts, you should use the latest released version of Solidity. This is because breaking changes as well as new features and bug fixes are introduced regularly.” In more detail, the contributions of this paper are:

(1) An executable formal semantics, using monads to capture state, of the *core of Solidity* as conservative embedding into Isabelle/HOL.
(2) A formalisation of *contracts* identified by addresses and consisting of member variables and methods.
(3) A formal model of method invocations, both within the same contract (internal method invocation) and to arbitrary contracts on the blockchain (external method invocation).
(4) A case study demonstrating how the formal semantics can be used, within Isabelle/HOL, to verify the correctness of a smart contract.

The semantics as well as the case studies are completely mechanised in Isabelle/HOL and available in the archive of formal proofs [39].

An older version of the semantics (contribution 1) is described in a previously published conference paper [37]. This journal article presents a significantly updated version of the semantics with several extensions. Compared to [37], the semantics has been completely redesigned using monads. In addition, we added support for contracts as well as internal and external method invocations.

The remainder of the paper is structured as follows: In Sect. 2 we provide some background to our work. In particular, we discuss the notion of finite maps and state monads which build the foundation for our semantics. Sect. 3 presents our formalization of the storage model of Solidity. In Sect. 4 we present our formalization of environments and variable declarations. We then present our formalization of Solidity expressions (Sect. 5) and statements (Sect. 6). In Sect. 7 we discuss how the gas model of Solidity can be leveraged to ensure termination of the semantic functions. To this end, we describe a corresponding measure function and how it can be used to prove termination. In Sect. 8 we describe a case study showing how to verify correctness of a simple banking contract. We discuss our experience in formalising Solidity and limitations of our approach (Sect. 9) and related work (Sect. 10). Sect. 11 concludes the paper with a discussion of future work.

Our formalization is based on the official Solidity v0.5.16 language specification[1]. All the Isabelle theory files related to this paper are publicly available[2] and we will refer to them throughout the paper. To this end we will use

Types

to refer to definition `Types` in the theory `Valuetypes.thy`. The theory is usually clear from the context and the Isabelle symbol is a clickable link to the corresponding part of the formalization.

2 BACKGROUND

Our formalization is based on higher-order logic using inductive data types [9]. To this end, we use **bold** font for types and *italics* for type constructors. We shall also use

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

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

2.1 Finite maps

Our formalization is based on the concept of finite maps. If **A** and **B** are types, we use

$$\mathbf{A} \Rightarrow \mathbf{B}$$

[1] https://docs.soliditylang.org/en/v0.5.16/
[2] https://github.com/dmarmsoler/isabelle-solidity-deep

to denote a finite map from $\mathbf{A}$ to $\mathbf{B}$. Such a map associates *at most* one element of type $\mathbf{B}$ with each element of type $\mathbf{A}$. Given a map $m\colon \mathbf{A} \Rightarrow \mathbf{B}$ we denote with $m(a) \in \mathbf{B}_\perp$ the unique element of $\mathbf{B}$ associated with $a \in \mathbf{A}$ or $\perp$ in the case no such element exists. The empty map is denoted *fmempty* and with $m[a \mapsto b]$ we associate element $b \in \mathbf{B}$ with element $a \in \mathbf{A}$ in mapping m.

2.2 State monads

To model stateful computations in a purely functional world such as HOL, we use monads, a widely used concept for modelling stateful computations, which is also frequently used in Isabelle/HOL for this purpose (e.g., [14, 51]). Actually, we make use of the monad syntax provided by Isabelle/HOL (in the standard theory `Monad_Syntax.thy`). In more detail, our semantics is defined using the concept of a state monad [17, 54][3]. To this end we first define a result type as follows:

$$\mathbf{Result}(\mathbf{N}, \mathbf{E}) \quad ::= \quad \mathit{Normal}(\mathbf{N}) \mid \mathit{Exception}(\mathbf{E}) \qquad (\texttt{result})$$

The type **Result** is defined over two type parameters **N** and **E** to denote the type for normal and exceptional return values, respectively. Thus, $\mathit{Normal}(s)$ represents a normal return with value s while $\mathit{Exception}(e)$ represents an exceptional return with exception e.

We can then define our monad as follows:

$$\mathbf{State\_Monad}(\mathbf{A}, \mathbf{E}, \mathbf{S}) \overset{\text{def}}{=} \mathbf{S} \rightarrow \mathbf{Result}(\mathbf{A} \times \mathbf{S}, \mathbf{E}) \qquad (\texttt{state\_monad})$$

The monad requires three type parameters:

- A type **A** for return values
- A type **S** for states
- A type **E** for exceptions

Such a monad is then supposed to update the state and return an element of type **A** or return an exception of type **E**.

2.2.1 *Bind.*

The bind operator $f \gg\!= g\ (s)$ allows to combine a monad $f\colon \mathbf{State\_Monad}(\mathbf{A}, \mathbf{E}, \mathbf{S})$ with a function $g\colon \mathbf{A} \rightarrow \mathbf{State\_Monad}(\mathbf{B}, \mathbf{E}, \mathbf{S})$:

$$f \gg\!= g\ (s) \overset{\text{def}}{=} \begin{cases} g(a)(s') & \mathit{if}\ f(s) = \mathit{Normal}(a, s') \\ \mathit{Exception}(e) & \mathit{if}\ f(s) = \mathit{Exception}(e) \end{cases} \qquad (\texttt{bind})$$

In addition, we shall use the following constants to denote normal and erroneous returns:

$$\mathit{return}(a) \overset{\text{def}}{=} \lambda s.\ \mathit{Normal}(a, s) \qquad (\texttt{return})$$

$$\mathit{throw}(e) \overset{\text{def}}{=} \lambda s.\ \mathit{Exception}(e) \qquad (\texttt{throw})$$

Note that our monad satisfies the usual laws required for monads. First, return is absorbed on the left of a bind, applying the return value directly.

Lemma 2.1 (`return_bind`). *For function $f\colon \mathbf{A} \rightarrow \mathbf{State\_Monad}(\mathbf{A}, \mathbf{E}, \mathbf{S})$ and an $a \in \mathbf{A}$, we have:*

$$(\mathit{return}(a) \gg\!= f) = f(a)$$

Moreover, return is absorbed on the right of a bind.

Lemma 2.2 (`bind_return`). *For a monad $m\colon \mathbf{State\_Monad}(\mathbf{A}, \mathbf{E}, \mathbf{S})$, we have:*

$$(m \gg\!= \mathit{return}) = m$$

Finally, bind is associative.

[3]State monads are formalized in theory `StateMonad.thy`

LEMMA 2.3 (bind_assoc). *For a monad* m: $\mathbf{State\_Monad}(\mathbf{A}, \mathbf{E}, \mathbf{S})$, *and functions* $f\colon \mathbf{A} \longrightarrow \mathbf{State\_Monad}(\mathbf{B}, \mathbf{E}, \mathbf{S})$ *and* $g\colon \mathbf{B} \longrightarrow \mathbf{State\_Monad}(\mathbf{C}, \mathbf{E}, \mathbf{S})$*, we have:*

$$(m \gg\!= f) \gg\!= g = m \gg\!=(\lambda x.\ f(x) \gg\!= g)$$

To combine multiple monads we will often use the familiar do-notation [4]. Then,

$$\mathbf{do}\left[\begin{array}{l} a \longleftarrow return(1) \\ return(a) \end{array}\right.$$

denotes the monad given by

$$return(1) \gg\!=(\lambda x.\ return(x))$$

2.2.2 Manipulating monads. We shall use the following functions to retrieve or change the state of a monad:

$$get(s) \overset{\text{def}}{=} Normal(s, s) \qquad (\texttt{get})$$

$$put(s, \_) \overset{\text{def}}{=} Normal((), s) \qquad (\texttt{put})$$

In addition, we shall use the following functions to modify a monad's state or simply apply a function to it:

$$modify(f) \overset{\text{def}}{=} get \gg\!=(\lambda s.\ put(f(s))) \qquad (\texttt{modify})$$

$$apply(f) \overset{\text{def}}{=} get \gg\!=(\lambda s.\ return(f(s))) \qquad (\texttt{applyf})$$

2.2.3 Assertions. The assert monad allows to specify a state predicate which determines an erroneous situation:

$$assert(x, c, m) \overset{\text{def}}{=} \lambda s. \begin{cases} throw(x) & \textit{if } c(s) \\ m(s) & \textit{if } \neg c(s) \end{cases} \qquad (\texttt{assert})$$

We shall usually use the following notation to denote an assert monad:

$$assert\ x\left[\begin{array}{l} c \\ m \end{array}\right.$$

2.2.4 Options. The option monad allows us to use option types with monads:

$$\text{option}(x, f, g) \overset{\text{def}}{=} \lambda s. \begin{cases} g(y)(s) & \textit{if } f(s) = y_{\perp} \\ Exception(x) & \textit{if } f(s) = \perp \end{cases} \qquad (\texttt{option})$$

Again, we often use the following notation to denote an option monad:

$$option\ x\left[\begin{array}{l} f \\ g \end{array}\right.$$

3 STORES AND STATES

In the following, we describe the storage model for our formalization. To this end, we first discuss the formalization of the basic types of data which can be stored (Sect. 3.1). Then, we discuss the different types of stores involved in Solidity (Sect. 3.2). Finally, we discuss the notion of a state in a Solidity program (Sect. 3.3).

[4]formalized in https://www.isa-afp.org/sessions/hol-library/#Monad_Syntax.html

3.1 Value types

Value types are formalized in theory `Valuetypes.thy`. Our version of Solidity supports four different basic data types, called *value types*:

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

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

In Solidity, raw data is encoded in hexadecimal format, however, to simplify the computation of locations for reference types (as discussed in more detail in Sect. 3.2), we use strings to represent raw data in our model. Thus, type **Valuetype** is actually just a synonym for type string, and it is used to represent the data of value types in the store. In addition, we shall write $\lfloor v \rfloor$ and $\lceil v \rceil$ to convert the value v of a basic data type to and from a string representation, respectively.

3.1.1 *Overflows.*

Converting an arbitrary integer to a fixed-width bit representation can result in an overflow or underflow. Consider, for example, the following two Solidity statements:

```
1 assert(uint8(500) == uint8(244)); //true
2 assert(int8(200) ==  int8(-56));  //true
```

In Line 1, we create an 8-bit unsigned integer from number 500. However, since the maximum size of such an integer is 256 we get an overflow which results in $500 - 256 = 244$. A similar situation arises with signed integers as shown in Line 2. This time, however, the size is from -128 to $+127$ which is why a value of 200 results in $-128 + (200 - 128) = -56$.

Thus, we define two functions *createUInt* and *createSInt* to convert an arbitrary number to a corresponding unsigned or signed integer representation of a certain size. The definition of *createUInt* is simple:

$$\mathit{createUInt}\colon \textbf{Nat} \times \textbf{Int} \longrightarrow \textbf{Valuetype} \qquad (\texttt{createUInt})$$

$$\mathit{createUInt}(b, v) \overset{\text{def}}{=} \left\lfloor v \bmod 2^{b-1} \right\rfloor$$

where "$x \bmod y$" denotes the non-negative remainder when dividing x by y.

The definition of *createSInt* is a bit more complex since we need to distinguish between positive and negative values:

$$\mathit{createSInt}\colon \textbf{Nat} \times \textbf{Int} \longrightarrow \textbf{Valuetype} \qquad (\texttt{createSInt})$$

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

Essentially, the functions can be used to create a representation of a given number which fits into a certain bit size. For example, constructing an unsigned 8-bit integer of 500 results in an overflow:

$$\mathit{createUInt}(8, 500) = \text{“}\mathit{244}\text{”}$$

Similarly we get an overflow if we create a signed 8-bit integer from 200: $\mathit{createSInt}(8, 200) = \text{“}\mathit{-56}\text{”}$.

Sometimes we need to convert a value between types. To this end we define function

$$\mathit{convert}\colon \textbf{Types} \times \textbf{Types} \times \textbf{Valuetype} \longrightarrow (\textbf{Valuetype} \times \textbf{Types})_{\bot} \qquad (\texttt{convert})$$

using usual pattern-matching notation as follows:

$$convert(\mathit{TSInt}(b_1), \mathit{TSInt}(b_2), v) \overset{\text{def}}{=} \begin{cases} (v, \mathit{TSInt}(b_2))_\perp & \text{if } b_1 \leq b_2 \\ \perp & \text{otherwise} \end{cases}$$

$$convert(\mathit{TUInt}(b_1), \mathit{TUInt}(b_2), v) \overset{\text{def}}{=} \begin{cases} (v, \mathit{TUInt}(b_2))_\perp & \text{if } b_1 \leq b_2 \\ \perp & \text{otherwise} \end{cases}$$

$$convert(\mathit{TUInt}(b_1), \mathit{TSInt}(b_2), v) \overset{\text{def}}{=} \begin{cases} (v, \mathit{TSInt}(b_2))_\perp & \text{if } b_1 < b_2 \\ \perp & \text{otherwise} \end{cases}$$

$$convert(\mathit{TBool}, \mathit{TBool}, v) \overset{\text{def}}{=} (v, \mathit{TBool})_\perp$$

$$convert(\mathit{TAddr}, \mathit{TAddr}, v) \overset{\text{def}}{=} (v, \mathit{TAddr})_\perp$$

$$convert(\_, \_, \_) \overset{\text{def}}{=} \perp$$

If the types are compatible then the function simply returns the original value with the new type. However, if the types are not compatible the function returns an error value $\perp$. In particular, we can convert a signed/unsigned integer to another signed/unsigned integer of equal or larger bit size. In addition, we can convert an unsigned integer to a signed integer only if the size of the signed one is strictly larger than the size of the unsigned one. Finally, we can never convert a signed integer to an unsigned one.

3.1.2 Arithmetic. We can then define functions to lift basic arithmetic and relational operations to corresponding operations over signed and unsigned integers of various sizes. According to the current specification of Solidity, adding two integers of the same type is always possible but results in a new integer of the size of the larger one. Adding integers of different type is only possible if the size of the signed integer is strictly greater than the one of the unsigned one, in which case the result is always a signed integer with the size of the signed one. Moreover, the result of adding two numbers might not fit into the corresponding result type in which case an overflow occurs.

Consider, for example, the following Solidity statements:

```
1 assert(uint8(50) + uint8(50) == uint8(100));         // true
2 assert(uint16(100) + int16(32700));                  // compiler error
3 assert(uint8(200) + int16(32600) == int16(-32736)); // true
```

In Line 1 we add two 8-bit unsigned integers of 50 which results in a corresponding 8-bit unsigned integer of 100. However, in Line 2 we try to add a 16-bit unsigned integer with a 16-bit signed integer which results in a compiler error since the two types are not compatible. Finally, in Line 3 we add an 8-bit unsigned integer of 200 with a 16-bit signed integer of 32 600 which results in a 16-bit signed integer of 32 800. However, since this does not fit in the result type we get again an overflow which gives us −32 736 as final value.

In the following, we define function

$$add\colon \mathbf{Types} \times \mathbf{Types} \times \mathbf{Valuetype} \times \mathbf{Valuetype} \longrightarrow (\mathbf{Valuetype} \times \mathbf{Types})_\perp \qquad \text{(add)}$$

as follows:

$$add\,(\mathit{TUInt}(b_l), \mathit{TUInt}(b_r), v_l, v_r) \overset{\text{def}}{=} createU\,(max(b_l, b_r), \lceil v_l \rceil + \lceil v_r \rceil)$$

$$add\,(\mathit{TSInt}(b_l), \mathit{TSInt}(b_r), v_l, v_r) \overset{\text{def}}{=} createS\,(max(b_l, b_r), \lceil v_l \rceil + \lceil v_r \rceil)$$

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

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

$$add\,(\_, \_, \_, \_) \overset{\text{def}}{=} \perp$$

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

We can now use our function to evaluate the examples above:

$$add(TUInt(8), TUInt(8), \text{“}50\text{”}, \text{“}50\text{”}) = (\text{“}100\text{”}, TUInt(8))_\perp$$
$$add(TUInt(16), TSInt(16), \text{“}100\text{”}, \text{“}32700\text{”}) = \perp$$
$$add(TUInt(8), TSInt(16), \text{“}200\text{”}, \text{“}32600\text{”}) = (\text{“}{-32736}\text{”}, TSInt(16))_\perp$$

Similar definitions are provided for the remaining arithmetic and relational operators: subtraction (sub), equality (equal), less than (less), and less or equal (leq).

3.1.3 Logical operations. Finally, we can lift basic logical operators such as

$$vtand\colon \mathbf{Types} \times \mathbf{Types} \times \mathbf{Valuetype} \times \mathbf{Valuetype} \to (\mathbf{Valuetype} \times \mathbf{Types})_\perp \qquad (\texttt{vtand})$$

as follows:

$$vtand(TBool, TBool, a, b) \overset{\text{def}}{=} \begin{cases} (\lfloor True \rfloor, TBool)_\perp & \text{if } a = \lfloor True \rfloor \wedge b = \lfloor True \rfloor \\ (\lfloor False \rfloor, TBool)_\perp & \text{otherwise} \end{cases}$$

$$vtand(\_, \_, \_, \_) \overset{\text{def}}{=} \perp$$

Again, a similar definition is provided for logical disjunction denoted *vtor* (vtor).

3.2 Stores and Reference Types

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

$$\mathbf{Store}(v) \quad ::= \quad (\mathbf{Loc} \Rightarrow v) \times \mathbf{Nat} \qquad (\texttt{Store})$$

It 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 with $mapping(s)$ the mapping of a store s while we use $toploc(s)$ to denote its top location. We can then define the following functions to create an empty store and access/update elements of a store:

$$emptyStore \overset{\text{def}}{=} (fmempty, 0) \qquad (\texttt{emptyStore})$$
$$accessStore(l, s) \overset{\text{def}}{=} mapping(s)(l) \qquad (\texttt{accessStore})$$
$$updateStore(l, v, s) \overset{\text{def}}{=} (mapping(s)[l \mapsto v], toploc(s)) \qquad (\texttt{updateStore})$$

Moreover, we can define a function to allocate new memory and return the new top location as follows:

$$allocate(s) \overset{\text{def}}{=} (\lfloor ntop \rfloor, (mapping(s), ntop)), \text{ where } ntop = toploc(s) + 1 \qquad (\texttt{allocate})$$

Finally, we define a function to add a new element to the top of a store:

$$push(v, s) \overset{\text{def}}{=} (allocate(updateStore(\lfloor toploc(s) \rfloor, v, s)))_2 \qquad \text{(push)}$$

where $(x)_n$ denotes the projection to the nth component of a tuple x.

3.2.1 Computing storage locations. The way Solidity computes storage locations for reference types is a bit special and thus worth a closer look. In Solidity the storage location for elements of a reference type are calculated by combining the address of the reference type variable with the corresponding index and hashing the result using the Keccak hash function [10]. Consider, for example, the simple contract in Listing 1 and assume that variable `balances` refers to stack location $0x \ldots 0047e7ef24$. Assume that we call deposit with value $0x \ldots 5f56beddC4$ for address a. Now the storage location modified by the assignment in Line 6 is *keccak256*(“. . . *5f56beddC4* . . . *0047e7ef24*”).

Listing 1: Solidity contract demonstrating the calculation of storage locations.

```
1 contract EtherStore {
2   mapping(address => uint) public balances;
3
4   function deposit(address a, uint256 v) public {
5     // balances[a] refers to location keccak256(a+balances)
6     balances[a] = v;
7   }
8 }
```

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

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

An important property of our representation of a hash function is that a given value indeed *uniquely* identifies location and index of a given element.

Proposition 3.1 (hash_injective). *Assuming that* $h(l, i) = h(l', i')$ *and* “.” $\notin i$ *and* “.” $\notin i'$, *then we have*

$$l = l' \quad and \quad i = i'$$

Thus, if we refrain from using dots in indices, we can be sure that our hash function is collision resistant, i.e., that different locations or indices indeed produce different hash values. Note, however, that we are not allowed to use dots in index names as otherwise collisions can occur as demonstrated by the following example:

$$h(\text{“}1.0\text{”}, \text{“}2\text{”}) = \text{“}2.1.0\text{”} = h(\text{“}0\text{”}, \text{“}2.1\text{”})$$

Listing 2: Solidity contract demonstrating different types of storage.

```
 1 contract Example {
 2   mapping(address => uint256) myMapping; //storage map
 3
 4   uint8[2][3] myStorageArray; //storage array
 5
 6   function example(uint8[2] calldata myCDArray) external {  //calldata array
 7     uint8[2] storage myPointer = myStorageArray[1]; //storage pointer
 8
 9     uint8[2] memory myMemoryArray; //memory array
10   }
11 }
```

3.2.2 *Types of Stores.*

Solidity supports four different types of stores: stack, memory, calldata, and storage. Consider, for example, the simple contract in Listing 2. The mapping declared in Line 2 and the array declared in Line 4 are contract variables which are kept in storage, a persistent store on the blockchain. The array declared in Line 6 as part of the formal parameters of method `example` is kept in calldata, a volatile store which is mainly used to pass data for method invocations. The storage array declared in Line 7 and the memory array declared in Line 9 are both pointers which are kept on the stack. The first one is a pointer to the storage location where the second array of `myStorageArray` is located. The second one is a pointer to a newly allocated array in memory, a volatile store for temporary data storage.

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 keep four different types of values:

$$\begin{aligned}\textbf{Stackvalue} \quad &::= \quad \textit{Simple}(\textbf{Valuetype}) \mid \textit{Memptr}(\textbf{Loc}) \\ &\quad\ \mid \textit{CDptr}(\textbf{Loc}) \mid \textit{Stoptr}(\textbf{Loc}) && (\texttt{Stackvalue}) \\ \textbf{Stack} \quad &::= \quad \textbf{Store}(\textbf{Stackvalue}) && (\texttt{Stack})\end{aligned}$$

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:

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

The internal organization of the three stores differs 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 keep two different types of values:

$$\begin{aligned}\textbf{Memoryvalue} \quad &::= \quad \textit{Value}(\textbf{Valuetype}) \mid \textit{Pointer}(\textbf{Loc}) && (\texttt{Memoryvalue}) \\ \textbf{Memory} \quad &::= \quad \textbf{Store}(\textbf{Memoryvalue}) && (\texttt{MemoryT}) \\ \textbf{Calldata} \quad &::= \quad \textbf{Memory} && (\texttt{CalldataT})\end{aligned}$$

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

$$\textbf{Storage} \quad ::= \quad \textbf{Loc} \Rightarrow \textbf{Valuetype} \qquad (\texttt{StorageT})$$

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

$$\mathit{ival}\colon \textbf{Types} \longrightarrow \textbf{Valuetype} \qquad (\texttt{ival})$$

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

$$\mathit{accessStorage}\colon \textbf{Types} \times \textbf{Loc} \times \textbf{Storage} \longrightarrow \textbf{Valuetype} \qquad (\texttt{accessStorage})$$

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

3.2.3 Copying data. Our model also provides functions to copy reference types. To this end we use a higher-order function

$$\mathit{iter}'\colon (\textbf{Int} \longrightarrow a \longrightarrow a_\perp) \longrightarrow a \longrightarrow \textbf{Int} \longrightarrow a_\perp \qquad (\texttt{iter'})$$

such that $\mathit{iter}'(f, v, x)$ executes function f on value v and then passes the outcome on to another execution of f until f was executed x times. Note that iter' is strict in the sense that it returns $\perp$ if one of the applications of f returns $\perp$.

Memory. Now we can define the following function to copy x elements of type t from location l_s of memory m_s to location l_d of memory m_d:

$$cp^m_m\colon \textbf{Loc} \times \textbf{Loc} \times \textbf{Int} \times \textbf{MTypes} \times \textbf{Memory} \times \textbf{Memory} \longrightarrow \textbf{Memory}_\perp \qquad (\texttt{cpm2m})$$

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

where

$$\mathit{cprec}^m_m(l_s, l_d, \mathit{MTArray}(x, t), m_s, m_d) \overset{\text{def}}{=} \begin{cases} \mathit{iter}'(go, m, x), & \\ & \text{if } \mathit{accessStore}(l_s, m_s) = \mathit{Pointer}(l)_\perp \\ \perp, & \text{otherwise} \end{cases} \tag{1}$$

$$\mathit{cprec}^m_m(l_s, l_d, \mathit{MTValue}(t), m_s, m_d) \overset{\text{def}}{=} \begin{cases} \mathit{updateStore}(l_d, \mathit{Value}(v), m_d), & \\ & \text{if } \mathit{accessStore}(l_s, m_s) = \mathit{Value}(v)_\perp \\ \perp, & \text{otherwise} \end{cases} \tag{2}$$

with $go = \lambda i, m'.\ \mathit{cprec}^m_m(h(l_s, \lfloor i \rfloor), h(l_d, \lfloor i \rfloor), t, m_s, m')$ and $m = \mathit{updateStore}(l_d, \mathit{Pointer}(l_d), m_d)$.

Remember that, in memory, reference types are stored using pointer structures and function cp^m_m needs to traverse the pointer structure in the source and recreate the structure in the destination. Thus, when copying an array in Equation 1 we first need to look up the corresponding pointer from the source memory location l_s and create a new pointer in the destination memory location l_d. Value types are simply copied as shown in Equation 2.

Consider, for example, a memory structure $\mathit{mymemory}$: **Memory** defined as follows:

- "*0.0*" $\mapsto$ *Pointer*("*0.0*")
- "*0.0.0*" $\mapsto$ *Value*("*True*")
- "*1.0.0*" $\mapsto$ *Value*("*False*")
- "*1.0*" $\mapsto$ *Pointer*("*1.0*")
- "*0.1.0*" $\mapsto$ *Value*("*True*")
- "*1.1.0*" $\mapsto$ *Value*("*False*")

Now assume we want to copy the second array of *mymemory* to location "*5*" of a new, empty memory and execute:

$$cp^m_m(\text{“}\mathit{1.0}\text{”}, \text{“}\mathit{5}\text{”}, 2, \mathit{MTValue}(\mathit{TBool}), \mathit{mymemory}, \mathit{emptyStore})$$

The result would be a memory with the following mapping:

- "*0.5*" $\mapsto$ *Value*("*True*")
- "*1.5*" $\mapsto$ *Value*("*False*")

Storage. We can define a similar function to copy reference types from one location to another on storage:

$$cp^s_s \colon \mathbf{Loc} \times \mathbf{Loc} \times \mathbf{Int} \times \mathbf{STypes} \times \mathbf{Storage} \longrightarrow \mathbf{Storage}_\perp \qquad \text{(copy)}$$

$$cp^s_s(l_s, l_d, x, t, sto) \overset{\text{def}}{=} iter'(\lambda i, s'.\ cprec^s_s(h(l_s, \lfloor i \rfloor), h(l_d, \lfloor i \rfloor), t, s'), sto, x)$$

where

$$cprec^s_s(l_s, l_d, \mathit{STArray}(x, t), sto) \overset{\text{def}}{=} iter'(\lambda i, s'.\ cprec^s_s(h(l_s, \lfloor i \rfloor), h(l_d, \lfloor i \rfloor), t, s'), sto, x) \tag{3}$$

$$cprec^s_s(l_s, l_d, \mathit{STValue}(t), sto) \overset{\text{def}}{=} sto[l_d \mapsto \mathit{accessStorage}(t, l_s, sto)]_\perp \tag{4}$$

$$cprec^s_s(\_, \_, \mathit{STMap}(\_, \_), \_) \overset{\text{def}}{=} \perp \tag{5}$$

Remember that unlike for memory, reference types are not stored using pointer structures in storage. Thus, we do not need to access storage when computing the storage location of an array element as shown in Equation 3. Moreover, observe that we use function *accessStorage* to lookup elements in Equation 4 which provides default values for elements which are not initialized. Finally, storage can keep mappings which is addressed by Equation 5. However, since Solidity does not allow enumerating mappings we simply return an error value for this case.

Consider for example a storage *mystorage*: **Storage** defined by the following mapping:

- "*0.0.0*" $\mapsto$ "*False*"
- "*1.1.0*" $\mapsto$ "*True*"

Now lets again try to copy the second array of the storage to location "*5*" by executing

$$cp^s_s(\text{“}\mathit{1.0}\text{”}, \text{“}\mathit{5}\text{”}, 2, \mathit{STValue}(\mathit{TBool}), \mathit{mystorage})$$

which results in the following, modified storage:

- "*0.0.0*" $\mapsto$ "*False*"
- "*1.1.0*" $\mapsto$ "*True*"
- "*0.5*" $\mapsto$ "*False*"
- "*1.5*" $\mapsto$ "*True*"

Note that the original storage was modified by adding two new elements: The one in location "*1.5*" is the copy of element "*1.1.0*" whereas the element at location "*0.5*" is initialized with the default value for booleans since the corresponding location "*0.1.0*" was not defined in the original storage.

3.2.4 Initializing Memory. When declaring new memory arrays we need to initialize them by creating the corresponding pointer structure. To this end, our model provides a function to initialize memory arrays. To specify it, we use a higher-order function

$$iter \colon (\mathbf{Int} \longrightarrow a \longrightarrow a) \longrightarrow a \longrightarrow \mathbf{Int} \longrightarrow a \qquad (\texttt{iter})$$

such that $iter(f, v, x)$ executes function f on value v and then passes the outcome on to another execution of f until f was executed x times. Note that *iter* is similar to function *iter'* defined above except that it does not support errors.

We can then define the following function to initialize a memory array:

$$\mathit{minit}\colon \mathbf{Int} \times \mathbf{MTypes} \times \mathbf{Memory} \to \mathbf{Memory} \qquad (\texttt{minit})$$

$$\mathit{minit}(i, t, m) \overset{\text{def}}{=} (\mathit{allocate}(\mathit{iter}(\lambda i', m'.\ \mathit{initrec}(h(\lfloor \mathit{top}(m) \rfloor, \lfloor i' \rfloor), t, m'), m, i)))_2$$

where

$$\mathit{initrec}(\mathit{loc}, \mathit{MTArray}(i, t)) \overset{\text{def}}{=} \mathit{iter}(\lambda i', m'.\ \mathit{initrec}(h(\mathit{loc}, \lfloor i' \rfloor), t, m'), m'', i) \tag{6}$$

$$\mathit{initrec}(\mathit{loc}, \mathit{Value}(t)) \overset{\text{def}}{=} \mathit{updateStore}(\mathit{loc}, (\mathit{Value}(\mathit{ival}(t)))) \tag{7}$$

with $m'' = \mathit{updateStore}(\mathit{loc}, \mathit{Pointer}(\mathit{loc}), m)$.

To initialize the array we first create the corresponding pointer structure in Equation 6. In addition, we initialize the elements using their default value in Equation 7.

Let's assume, for example, we want to initialize a fresh memory with a two-dimensional boolean array. Then, we could use

$$\mathit{minit}(2, \mathit{MTArray}(2, \mathit{MTValue}(\mathit{TBool})), \mathit{emptyStore})$$

The result would be store $(\mu, 1)$ with μ containing the following entries:

- "*0.0*" $\mapsto$ *Pointer*("*0.0*")
- "*0.0.0*" $\mapsto$ *Value*("*False*")
- "*1.0.0*" $\mapsto$ *Value*("*False*")
- "*1.0*" $\mapsto$ *Pointer*("*1.0*")
- "*0.1.0*" $\mapsto$ *Value*("*False*")
- "*1.1.0*" $\mapsto$ *Value*("*False*")

The array is stored at location "*0*" which is why the top location is set to one. The first dimension of the array contains two entries: the first is located at "*0.0*" and the second one at "*1.0*". Both entries store pointers to the corresponding second dimension. In the first case, the pointer points to location "*0.0*" and the corresponding entries are located at "*0.0.0*" and "*1.0.0*", respectively. In the second case, the pointer points to location "*1.0*" and the corresponding entries are located at "*0.1.0*" and "*1.1.0*", respectively. Note that the primitive values are all initialized with the corresponding default value which for type boolean is "*False*".

3.2.5 *Inter-store copy.*

Often, we need to copy values from one type of store to another. Thus, our model provides additional functions to support inter-store copies. To specify them, we use again the higher-order function *iter′* introduced above:

$$\mathit{cp}^s_m\colon \mathbf{Loc} \times \mathbf{Loc} \times \mathbf{Int} \times \mathbf{STypes} \times \mathbf{Storage} \times \mathbf{Memory} \to \mathbf{Memory}_\bot \qquad (\texttt{cps2m})$$

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

where

$$\mathit{cprec}^s_m(l_s, l_m, \mathit{STArray}(i, t), s, m) \overset{\text{def}}{=} \mathit{iter}'(\lambda i', m'.\ \mathit{cprec}^s_m(h(l_s, \lfloor i' \rfloor), h(l_m, \lfloor i' \rfloor), t, s, m'), m'', i) \tag{8}$$

$$\mathit{cprec}^s_m(l_s, l_m, \mathit{STValue}(t), s, m) \overset{\text{def}}{=} \mathit{updateStore}(l_m, \mathit{Value}(\mathit{accessStorage}(t, l_s, s)), m)_\bot \tag{9}$$

$$\mathit{cprec}^s_m(l_s, l_m, \mathit{STMap}(t, t'), s, m) \overset{\text{def}}{=} \bot \tag{10}$$

with $m'' = \mathit{updateStore}(l_m, \mathit{Pointer}(l_m), m)$.

In Solidity, value types are just copied between stores which is reflected by Equation 9. For reference types, however, the situation is different. Mappings can only be kept in storage and not in memory which is why a mapping is never copied from storage to memory, and we just return $\bot$ for this case (Equation 10). Arrays, on the other hand, can be kept in both: storage and memory. As mentioned above, however, the way of storing them differs depending on the type of store: in storage, we just calculate the location of the elements of an array whereas in memory arrays are stored using a pointer structure. Thus, when copying arrays from storage to memory we need to create the corresponding pointer structure as shown by Equation 8.

Our model provides a similar function to copy data from memory to storage:

$$cp_s^m\colon \mathbf{Loc} \times \mathbf{Loc} \times \mathbf{Int} \times \mathbf{MTypes} \times \mathbf{Memory} \times \mathbf{Storage} \to \mathbf{Storage}_\perp \qquad (\texttt{cpm2s})$$

$$cp_s^m(l_m, l_s, i, t, m, s) \overset{\text{def}}{=} iter'(\lambda i', s'.\ cprec_s^m(h(l_m, \lfloor i' \rfloor), h(l_s, \lfloor i' \rfloor), t, m, s'), s, i)$$

where

$$cprec_s^m(l_m, l_s, MTArray(i,t), m, s) \overset{\text{def}}{=} \begin{cases} s'' & \text{if } accessStore(l_m, m) = Pointer(l)_\perp \\ \perp & \text{otherwise} \end{cases} \tag{11}$$

$$cprec_s^m(l_m, l_s, MTValue(t), m, s) \overset{\text{def}}{=} \begin{cases} s[l_s \mapsto v]_\perp & \text{if } accessStore(l_m, m) = Value(v)_\perp \\ \perp & \text{otherwise} \end{cases} \tag{12}$$

with $s'' = iter'\ (\lambda i', s'.\ cprec_s^m(h(l_m, \lfloor i' \rfloor), h(l_s, \lfloor i' \rfloor), t, m, s'), s, i)$.

Again value types are just copied between the stores which is reflected by Equation 12. However, since memory does not support mappings, we only have to consider one case for reference types. Note that this time we do not need to create a pointer structure for an array but rather we need to navigate such a structure to obtain all its elements (Equation 11).

Consider, for example a memory store $m = (\mu, 1)$, where μ contains the following mappings:

- *"0.0"* $\mapsto$ *Pointer("0.0")*
- *"0.0.0"* $\mapsto$ *Value("True")*
- *"1.0.0"* $\mapsto$ *Value("False")*
- *"1.0"* $\mapsto$ *Pointer("1.0")*
- *"0.1.0"* $\mapsto$ *Value("True")*
- *"1.1.0"* $\mapsto$ *Value("False")*

Note that m stores a two-dimensional boolean array at location 0.

We can now copy location *"0"* of the memory to location *"1"* of a newly created storage as follows:

$$cp_s^m(\text{"0"}, \text{"1"}, 2, MTArray(2, MTValue(Bool)), m, \emptyset)$$

Here, the third parameter specifies the length of the first array and the fourth parameter specifies the type of each entry of this array. The $\emptyset$ denotes a new, empty, storage.

The above function returns a storage s which contains the following mappings reflecting the entries of the memory array:

- *"0.0.1"* $\mapsto$ *"True"*
- *"1.0.1"* $\mapsto$ *"False"*
- *"0.1.1"* $\mapsto$ *"True"*
- *"1.1.1"* $\mapsto$ *"False"*

Note that the layout of the data has changed. In particular the storage version does not contain any pointers. In addition, the array is now stored at location *"1"* instead of *"0"*.

We can now copy the array from the storage back to a fresh memory to obtain a store equal to the original m:

$$cp_m^s(\text{"1"}, \text{"0"}, 2, STArray(2, STValue(Bool)), s, (allocate(emptyStore))_2)$$

Note, that the copy function does not change the top location automatically since it is not always the case that we allocate new memory when copying data. Thus, we needed to allocate a new slot before copying over the elements.

3.3 Accounts, Gas, and States

3.3.1 Accounts. Accounts are formalized in theory `Accounts.thy`. Each account is associated with an address in hexadecimal format. We model **Address** and **Balance** as strings and accounts as mappings from addresses to their balance:

$$\textbf{Accounts} \;::=\; \textbf{Address} \rightarrow \textbf{Valuetype} \qquad (\texttt{Accounts})$$

We can then define functions to modify balances. The function for adding some value to a certain account is defined as follows:

$$\mathit{addBalance}\colon \textbf{Address} \times \textbf{Balance} \times \textbf{Accounts} \longrightarrow \textbf{Accounts}_{\perp} \qquad (\texttt{addBalance})$$

$$\mathit{addBalance}(\mathit{ad}, \mathit{val}, \mathit{acc}) \overset{\text{def}}{=} \begin{cases} \mathit{acc}[\mathit{ad} \mapsto \lfloor v \rfloor]_{\perp} & \text{if } \lceil \mathit{val} \rceil \geq 0 \land \lceil \mathit{acc}(\mathit{ad}) \rceil + \lceil \mathit{val} \rceil < 2^{256} \\ \perp & \text{otherwise} \end{cases}$$

Note that the function fails whenever the value is negative or the new balance would produce an overflow. Otherwise, it simply adds the value to the balance of the given account. This is demonstrated by the following lemma:

Lemma 3.2 (`addBalance_add`). *Assuming $\mathit{addBalance}(\mathit{ad}, \mathit{val}, \mathit{acc}) = \mathit{acc'}_{\perp}$, then*

$$\lceil \mathit{acc'}(\mathit{ad}) \rceil = \lceil \mathit{acc}(\mathit{ad}) \rceil + \lceil \mathit{val} \rceil$$

Also note that, for the case it is successful, *addBalance* is monotonic:

Lemma 3.3 (`addBalance_mono`). *Assuming $\mathit{addBalance}(\mathit{ad}, \mathit{val}, \mathit{acc}) = \mathit{acc'}_{\perp}$, then*

$$\lceil \mathit{acc'}(\mathit{ad}) \rceil \geq \lceil \mathit{acc}(\mathit{ad}) \rceil$$

The function for removing funds from an account is denoted *subBalance* (`subBalance`) and defined similarly.

Finally, we can use the two functions to define a new function to transfer money from one account to another:

$$\mathit{transfer}\colon \textbf{Address} \times \textbf{Address} \times \textbf{Valuetype} \times \textbf{Accounts} \longrightarrow \textbf{Accounts}_{\perp} \qquad (\texttt{transfer})$$

$$\mathit{transfer}(\mathit{ads}, \mathit{adr}, \mathit{val}, \mathit{acc}) \overset{\text{def}}{=} \begin{cases} \mathit{addBalance}(\mathit{adr}, \mathit{val}, \mathit{acc'}) & \text{if } \mathit{subBalance}(\mathit{ads}, \mathit{val}, \mathit{acc}) = \mathit{acc'}_{\perp} \\ \perp & \text{if } \mathit{subBalance}(\mathit{ads}, \mathit{val}, \mathit{acc}) = \perp \end{cases}$$

Again, the function fails when either *subBalance* or *addBalance* fails. Otherwise, it just removes the value from the balance of one account and adds it to another one as shown by the following lemma:

Lemma 3.4 (Transfer). *Assuming $\mathit{transfer}(\mathit{ads}, \mathit{addr}, \mathit{val}, \mathit{acc}) = \mathit{acc'}_{\perp}$ and $\mathit{addr} \neq \mathit{ads}$, then*

$$\lceil \mathit{acc'}(\mathit{addr}) \rceil = \lceil \mathit{acc}(\mathit{addr}) \rceil + \lceil \mathit{val} \rceil \qquad (\texttt{transfer\_add})$$

and

$$\lceil \mathit{acc'}(\mathit{ads}) \rceil = \lceil \mathit{acc}(\mathit{ads}) \rceil - \lceil \mathit{val} \rceil \qquad (\texttt{transfer\_sub})$$

3.3.2 Gas. One interesting aspect of Solidity is that execution of expressions/statements is subject to fees, i.e., the execution consumes gas: if all gas is consumed, the execution terminates with an exception. To this end, we assume the existence of generic cost functions which provide the gas costs for executing a given expression/statement (see semantics of expressions/statements for more details on the cost functions). In addition, we define a special type of exception to be used with our monad:

$$\textbf{Exception} \;::=\; \mathit{Gas} \mid \mathit{Err} \qquad (\texttt{Ex})$$

Now we can define a simple monad to check availability of gas:

$$\mathit{gascheck}\colon (\textbf{State} \longrightarrow \textbf{Gas}) \longrightarrow \textbf{State\_Monad}((), \textbf{Exception}, \textbf{State}) \qquad (\texttt{gascheck})$$

$$\mathit{gascheck}(\mathit{check}) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} g \leftarrow \mathit{apply}(\mathit{check}) \\ \mathit{assert\ Gas} \left[\begin{array}{l} \lambda st.\ \mathit{gas}(st) \leq g \\ \mathit{modify}(\lambda st.\ \mathit{upGas}(\mathit{gas}(st) - g, \mathit{gas}(st))) \end{array} \right. \end{array} \right.$$

The *gascheck* monad uses an external function to determine the actual costs. Then it checks that enough gas is available and updates the state to deduce the costs. For the rest of this section, we use a generic gas model. We will later, in Sect. 7, constrain it slightly to ensure termination. A more detailed discussion on gas models is included in Sect. 9.

3.3.3 States. States are formalized in theory `Statements.thy`. They consist of the balances of the accounts as well as the configuration of the different stores and the remaining amount of gas:

$$\textbf{State} \quad ::= \quad \textbf{Accounts} \times \textbf{Stack} \times \textbf{Memory} \times (\textbf{Address} \Rightarrow \textbf{Storage}) \times \textbf{Nat} \qquad (\texttt{State})$$

Note that each address has its own storage. Later on we will see that a program can only access storage associated with its own address. In the following we shall use $acc(s)$, $sck(s)$, $mem(s)$, $sto(s)$, and $gas(s)$ to access the account, stack, memory, storage, and gas of a state s. Moreover, we shall use $upAcc(a, s)$, $upSck(k, s)$, $upMem(m, s)$, $upSto(t, s)$, and $upGas(g, s)$ to change account, stack, memory, storage, or gas, of a state s to a, k, m, t, or g, respectively.

4 ENVIRONMENTS AND DECLARATIONS

Variables are always interpreted with respect to an environment which assigns them types and values. In the following, we describe our formalization of environments. To this end, we first introduce the definition of an environment and some useful functions to manipulate environments (Sect. 4.1). Then, we discuss the definition of a declaration function to support the declaration of new variables (Sect. 4.2). Finally, we discuss our notion of contract environment to store data belonging to a contract (Sect. 4.3). Environments and declarations are formalized in theory `Environment.thy`.

4.1 Environment

To this end, we introduce a new type **Identifier** (a synonym of type string) for variable names. Variables in Solidity can either be a stack reference or a storage reference and refer to either a valuetype or a complex data type in one of the stores.

$$\begin{array}{rcl} \textbf{Type} & ::= & \mathit{Value}(\textbf{Types}) \mid \mathit{Calldata}(\textbf{MTypes}) \\ & & \mid \mathit{Memory}(\textbf{MTypes}) \mid \mathit{Storage}(\textbf{STypes}) \qquad (\texttt{Type}) \\ \textbf{Denvalue} & ::= & \mathit{Stackloc}(\textbf{Loc}) \mid \mathit{Storeloc}(\textbf{Loc}) \qquad (\texttt{Denvalue}) \end{array}$$

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

$$\textbf{Environment} \quad ::= \quad \textbf{Address} \times \textbf{Address} \times \textbf{Valuetype} \times (\textbf{Identifier} \Rightarrow \textbf{Type} \times \textbf{Denvalue}) \qquad (\texttt{Environment})$$

We use $address(env)$, $sender(env)$, $svalue(env)$, and $denvalue(env)$ to denote the address, sender, obtained funds, and denvalue of an environment env. Moreover, we use $upAdd(a, env)$, $upSender(s, env)$, $upVal(v, env)$, and $upDV(d, env)$ to update the address, sender, funds, and denvalue of env with a, s, v, and d. Finally, we use $empty(s, s, v)$ to denote an empty environment with address s, sender s, and value v (`emptyEnv`).

Sometimes it is required to update an environment only if the identifier is not yet declared. To do so, we define the following function:

$$\mathit{updateEnvDup}\colon \textbf{Identifier} \times \textbf{Type} \times \textbf{Denvalue} \times \textbf{Environment} \to \textbf{Environment} \qquad (\texttt{updateEnvDup})$$

$$\mathit{updateEnvDup}(i, t, v, e) \overset{\text{def}}{=} \begin{cases} e & \text{if } \mathit{denvalue}(e)(i) \neq \bot \\ \mathit{upDV}(\mathit{denvalue}(e)[i \mapsto (t, v)], e) & \text{if } \mathit{denvalue}(e)(i) = \bot \end{cases}$$

Often, when declaring a new variable, we want to update environment and stack at the same time. To this end, we define the following function:

$$\mathit{append}\colon \textbf{Identifier} \times \textbf{Type} \times \textbf{Stackvalue} \times (\textbf{Stack} \times \textbf{Environment}) \to (\textbf{Stack} \times \textbf{Environment}) \qquad (\texttt{astack})$$

$$\mathit{append}(\mathit{id}, \mathit{tp}, \mathit{vl}, (\mathit{sk}, \mathit{ev})) \overset{\text{def}}{=} (\mathit{push}(\mathit{vl}, \mathit{sk}), \mathit{upDV}(\mathit{denvalue}(\mathit{ev})[\mathit{id} \mapsto (\mathit{tp}, \mathit{Stackloc}(\lfloor \mathit{toploc}(\mathit{sk}) \rfloor))], \mathit{ev}))$$

Function *append* adds a new element to the top of the stack and adds a pointer to the location to the corresponding entry of the identifier in the environment.

Example 4.1 (Append). Let's assume we want to add a new boolean variable with value "*True*" to an empty environment and stack:

$$\mathit{append}(\text{``}\mathit{id1}\text{''}, \mathit{Value}(\mathit{TBool}), \mathit{Simple}(\text{``}\mathit{True}\text{''}), (\mathit{emptyStore}, \mathit{empty})) = (\mathit{mystack}, \mathit{myenv})$$

This results in an updated stack and environment, respectively, containing the following entries:

myenv :
- "*id1*" $\mapsto$ $(\mathit{Value}(\mathit{TBool}), \mathit{Stackloc}(\text{``}\mathit{0}\text{''}))$

mystack :
- "*0*" $\mapsto$ $\mathit{Simple}(\text{``}\mathit{True}\text{''})$

In particular, *myenv* now contains a new entry for variable "*id1*" which points to stack location "*0*" which contains the actual element on the new stack *mystack*. Note that the top location of the stack was updated and $\mathit{toploc}(\mathit{mystack}) = 1$. To append another element we can now execute

$$\mathit{append}(\text{``}\mathit{id2}\text{''}, \mathit{Value}(\mathit{TBool}), \mathit{Simple}(\text{``}\mathit{False}\text{''}), (\mathit{mystack}, \mathit{myenv})) = (\mathit{mystack}', \mathit{myenv}')$$

The resulting environment and stack are as follows:

myenv′ :
- "*id1*" $\mapsto$ $(\mathit{Value}(\mathit{TBool}), \mathit{Stackloc}(\text{``}\mathit{0}\text{''}))$
- "*id2*" $\mapsto$ $(\mathit{Value}(\mathit{TBool}), \mathit{Stackloc}(\text{``}\mathit{1}\text{''}))$

mystack′ :
- "*0*" $\mapsto$ $\mathit{Simple}(\text{``}\mathit{True}\text{''})$
- "*1*" $\mapsto$ $\mathit{Simple}(\text{``}\mathit{False}\text{''})$

The environment now contains a new entry for variable "*id2*" which holds a reference to the added entry on location "*1*" of the new stack.

4.2 Declarations

To support the declaration of new variables, our model provides the following function:

$$\mathit{decl}\colon \textbf{Identifier} \times \textbf{Type} \times (\textbf{Stackvalue} \times \textbf{Type}_{\bot}) \times \textbf{Bool} \times \textbf{Calldata} \times \textbf{Memory} \times (\textbf{Address} \Rightarrow \textbf{Storage}) \times (\textbf{Calldata} \times \textbf{Memory} \times \textbf{Stack} \times \textbf{Environment}) \to (\textbf{Calldata} \times \textbf{Memory} \times \textbf{Stack} \times \textbf{Environment})_{\bot} \qquad (\texttt{decl})$$

We can use this function to declare a new variable as follows:

$$\mathit{decl}\ \mathit{id}\ \mathit{tp}\ \mathit{vl}\ \mathit{cp}\ \mathit{cd}\ \mathit{mem}\ \mathit{st}\ (c, m, k, e) \overset{\text{def}}{=} (c', m', k', e')$$

This updates calldata c, memory m, stack k, and environment e to declare a new variable *id* of type *tp*. *vl* is an optional initialization value. If it is $\bot$, the type's default value is taken. The copy flag *cp* indicates whether memory should be copied (from *mem* parameter) or not. Copying is required, for example, for external method calls. *cd* and *mem* refer to the original calldata and memory stores which are used as a source for copying.

In the following we discuss the definition of *decl*. We begin with the declaration of value type variables:

$$decl(i, Value(t), \bot, \_, \_, \_, \_, (c, m, k, e)) \overset{\text{def}}{=} return(append(i, Value(t), Simple(ival(t)), (k, e))) \tag{13}$$

$$decl(i, Value(t), (Simple(v), Value(t'))_\bot, \_, \_, \_, \_, (c, m, k, e)) \overset{\text{def}}{=} \mathbf{do}\left[\begin{array}{l}(v', t'') \leftarrow convert(t', t, v)\\ return(c, m, append(i, Value(t''), Simple(v'), (k, e)))\end{array}\right. \tag{14}$$

$$decl(\_, Value(\_), \_, \_, \_, \_, \_, \_) \overset{\text{def}}{=} \bot \tag{15}$$

Equation 13 captures the case in which we declare a new variable without an initial value. In such a case we use function *ival* to determine the default value of the corresponding type which is used to initialize the variable. In the case we provide a default value (Equation 14), we first convert the value to the type of the variable and if this conversion is successful we use the corresponding value to initialize the variable. Inconsistencies in the declaration are captured by Equation 15 in which case we simply return an error.

Next we discuss the declaration of variables for calldata. Calldata is only initialized once when a new method is called, and it cannot be modified later on. Thus, we only need to consider the cases in which a structure is copied completely:

$$decl(i, Calldata(MTArray(x, t)), (CDptr(p), \_)_\bot, True, cd, \_, \_, (c, m, k, e)) \overset{\text{def}}{=} \mathbf{do}\left[\begin{array}{l}c'' \leftarrow cp_m^m(p, l, x, t, cd, c')\\ return(c'', m, append(i, Calldata(MTArray(x, t)), CDptr(l), (k, e)))\end{array}\right. \tag{16}$$

where $c' = (allocate(c))_2$ and $l = \lfloor toploc(c) \rfloor$.

$$decl(i, Calldata(MTArray(x, t)), (Memptr(p), \_)_\bot, True, \_, mem, \_, (c, m, k, e)) \overset{\text{def}}{=} \mathbf{do}\left[\begin{array}{l}c'' \leftarrow cp_m^m(p, l, x, t, mem, c')\\ return(c'', m, append(i, Calldata(MTArray(x, t)), CDptr(l), (k, e)))\end{array}\right. \tag{17}$$

$$decl(i, Calldata(\_), \_, \_, \_, \_, \_, \_) \overset{\text{def}}{=} \bot \tag{18}$$

In particular, we have two possibilities: Either the source lies in calldata (Equation 16) or the source lies in memory (Equation 17). In both cases we use the function cp_m^m to copy the array and initialize the variable with a pointer to the newly created structure. All other cases are captured by Equation 18 and just throw an error to signal an inconsistency with the parameter values.

Next we discuss the declaration of arrays for memory stores. The first case deals with situations in which we do not provide an explicit initialization value:

$$decl(i, Memory(MTArray(x, t)), \bot, \_, \_, \_, \_, (c, m, k, e)) \overset{\text{def}}{=} return(c, minit(x, t, m), append(i, Memory(MTArray(x, t)), Memptr(\lfloor toploc(m) \rfloor), (k, e)))$$

In such a case we use function *minit* discussed earlier to create the corresponding pointer structure in memory and initialize the elements with their default value. Then we just create a pointer to the corresponding location.

The next two cases capture the situation in which we do have an explicit initialization value. In such a situation, the behaviour of a declaration depends on the value of the copy flag. The situation in which it is true is as follows:

$$decl(i, Memory(MTArray(x,t)), (Memptr(p), \_)_\bot, True, \_, mem, \_, (c,m,k,e)) \overset{\text{def}}{=}$$
$$\textbf{do}\left[\begin{array}{l} m' \leftarrow cp^m_m(p, \lfloor toploc(m) \rfloor, x, t, mem, (allocate(m))_2) \\ return(c, m', append(i, Memory(MTArray(x,t)), Memptr(\lfloor toploc(m) \rfloor), (k,e))) \end{array}\right. \tag{19}$$

For this case we use function cp^m_m discussed above to copy the complete array structure from one memory to another. Then, we simply create a new pointer to the location of the copied memory location to initialize the variable. If the copy flag is false, the behaviour is as follows:

$$decl(i, Memory(MTArray(x,t)), (Memptr(p), \_)_\bot, False, \_, \_, \_, (c,m,k,e)) \overset{\text{def}}{=}$$
$$return(c, m, append(i, Memory(MTArray(x,t)), Memptr(p), (k,e)))$$

Thus, in such a case we just create a new pointer to the original memory location to initialize the variable.

The remaining cases deal with the situation in which the array is stored in a different type of store. The case in which the array is stored in calldata is similar to the situation described above in which the copy flag is set to true:

$$decl(i, Memory(MTArray(x,t)), (CDptr(p), \_)_\bot, \_, cd, \_, \_, (c,m,k,e)) \overset{\text{def}}{=}$$
$$\textbf{do}\left[\begin{array}{l} m' \leftarrow cp^m_m(p, \lfloor toploc(m) \rfloor, x, t, cd, (allocate(m))_2) \\ return(c, m', append(i, Memory(MTArray(x,t)), Memptr(\lfloor toploc(m) \rfloor), (k,e))) \end{array}\right. \tag{20}$$

In particular, we first use function cp^m_m to copy the structure from calldata to memory and initialize the variable with a pointer to the newly created structure. The situation in which the array is stored in storage is defined as follows:

$$decl(i, Memory(MTArray(x,t)), (Stoptr(p), Storage(STArray(x',t')))_\bot, \_, \_, \_, s, (c,m,k,e)) \overset{\text{def}}{=}$$
$$\textbf{do}\left[\begin{array}{l} s' \leftarrow s(address(e)) \\ m'' \leftarrow cp^s_m(p, \lfloor toploc(m) \rfloor, x', t', s', (allocate(m))_2) \\ return(c, m'', append(i, Memory(MTArray(x,t)), Memptr(\lfloor toploc(m) \rfloor), (k,e))) \end{array}\right. \tag{21}$$

Again we use function cp^s_m to copy the structure from storage to memory and then initialize the variable with a pointer to the newly created structure. Note, however, that we first need to look up the private store of the executing contract.

The last case captures unexpected calls to *decl* and returns an error:

$$decl(\_, Memory(\_), \_, \_, \_, \_, \_, \_) \overset{\text{def}}{=} \bot$$

Finally, we discuss the declaration of storage variables:

$$decl(i, Storage(STArray(x,t)), (Stoptr(p), \_)_\bot, \_, \_, \_, \_, (c,m,k,e)) \overset{\text{def}}{=}$$
$$return(c, m, append(i, Storage(STArray(x,t)), Stoptr(p), (k,e))) \tag{22}$$

$$decl(i, Storage(STMap(t,t')), (Stoptr(p), \_)_\bot, \_, \_, \_, \_, (c,m,k,e)) \overset{\text{def}}{=}$$

$$return(c, m, append(i, Storage(STMap(t, t')), Stoptr(p), (k, e))) \tag{23}$$

$$decl(\_, Storage(\_), \_, \_, \_, \_, \_, \_) \overset{\text{def}}{=} \bot \tag{24}$$

Equation 22 and Equation 23 capture the case in which the variable is initialised with a pointer to a storage array or storage map. Note that Solidity only allows for the declaration of storage pointers and not explicit data structures. Thus, the definitions above do not involve any copy operations. Moreover, storage pointers need to be initialized. Thus, the remaining cases in Equation 24 are not valid and throw an error.

An important property for declarations is that they do not modify the address, sender, and funds of the given environment:

Lemma 4.2 (`decl_address`). *Assuming*

$$decl(id, tp, vl, cp, cd, mm, st, (c, m, k, e)) = (c', m', k', e')_\bot$$

then

$$address(e') = address(e) \land sender(e') = sender(e) \land svalue(e') = svalue(e)$$

Proof. We need to consider all the possible cases from the definition of *decl*. Most of the cases are trivial and need no further discussion. An exception is the case for Equation 14 which needs to convert data from one type to another and thus requires an additional case distinction on the outcome of the conversion. Moreover, Equation 16, Equation 17, Equation 19, Equation 20, and Equation 21, copy data from one store to another and thus need an additional case distinction on the outcome of this copy process. □

4.3 Contract environments

In addition to an environment for variables we do also need an environment to store data and methods belonging to a contract. Contract environments are formalized in theory `Statements.thy` and consist of two parts:

$$\begin{array}{lll} \textbf{Member} & ::= & Method([\textbf{Identifier} \times \textbf{Type}] \times \textbf{S} \times \textbf{E}_\bot) \mid Var(\textbf{STypes}) \\ \textbf{Environment}_\textbf{P} & ::= & \textbf{Address} \Rightarrow (\textbf{Identifier} \Rightarrow \textbf{Member}) \times \textbf{S} \end{array}$$

(`Member`)
(`Environment`$_\text{P}$)

A contract is identified by an address and consists of members and a fallback method. Members can be either variables or methods. Variables keep the data elements of a contract which are of type **STypes**. Methods, on the other hand, consist of a body of type **S** and an optional return value of type **E**. These are the types for statements and expressions discussed below. A contract's fallback method is a statement of type **S** which is executed whenever the contract receives any funds as discussed further in the semantics of the transfer statement and external method calls.

To initialize a variable environment with the member variables of a contract we provide the following function:

$$init\colon (\textbf{Identifier} \Rightarrow \textbf{Member}) \times \textbf{Identifier} \times \textbf{Environment} \rightarrow \textbf{Environment}$$ (`init`)

$$init(ct, i, e) \overset{\text{def}}{=} \begin{cases} updateEnvDup(i, Storage(tp), Storeloc(i), e) & \text{if } ct(i) = Var(tp)_\bot \\ e & \text{otherwise} \end{cases}$$

The function simply updates a given variable environment by adding a new entry for a corresponding member variable. Note that the type of the identifier in the variable environment is set to $Storage(tp)$ where tp is the type of the member variable itself. Note that due to the use of *updateEnvDup*, *init* commutes:

$$init(ct, id, init(ct, id', ev)) = init(ct, id', init(ct, id, ev)) \qquad (\texttt{init\_commte})$$

This is an important property which allows us to fold *init* over a set to support the initialization of multiple variables.

5 EXPRESSIONS

Our subset of Solidity supports basic logical and arithmetic operations over signed and unsigned integers of various bit sizes. Moreover, we can reference variables and navigate complex data types. In addition, we can create addresses, query the balance of some address, or obtain the address of the currently executing contract. Finally, we can call internal and external functions, obtain the address which triggered the current execution, and obtain the value sent with it. The corresponding syntax of expressions is given by a data type **E** defined as follows:

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

where **String** denotes the type of strings and $[a]$ a list of elements of type a.

Expressions are formalized in theory `Statements.thy`. In the following, we discuss the semantics of expressions in more detail. To this end, we first discuss the notion of selectors (Sect. 5.1) and lookup functions (Sect. 5.2) to access the value of a variable. We then use these functions to define the semantics of each of our expressions (Sect. 5.3).

5.1 Selectors

To access the value of a reference type we define functions to look up the corresponding value in memory or storage. We first discuss the function

$$msel\colon \mathbf{Bool} \times \mathbf{MTypes} \times \mathbf{Loc} \times [\mathbf{E}] \times \mathbf{Environment}_p \times \mathbf{Environment} \times \mathbf{Calldata} \to \\ \mathbf{State\_Monad}(\mathbf{Loc} \times \mathbf{MTypes}, \mathbf{Exception}, \mathbf{State}) \qquad (\texttt{msel})$$

to lookup values in memory. The general idea is that

$$msel(mm, t, l, xs, e_p, env, cd)$$

looks up the index xs of an array of type t stored at location l. The flag mm can be used to change the source of the lookup from memory to calldata. The definition is as follows:

$$msel(\_, MTArray(al, t), loc, [x], e_p, env, cd) \overset{\text{def}}{=}$$
$$\mathbf{do}\left[\begin{array}{l}
kv \leftarrow expr(x, e_p, env, cd) \\
\lfloor\_\rfloor \begin{cases}
assert\ Err \left[\begin{array}{l} \lambda\_.\ \neg less(t', TUInt(256), v, \lfloor al \rfloor) = (\lfloor True \rfloor, TBool)_\perp \\ return(h(loc, v), t) \end{array}\right. & \text{if } kv = (Simple(v), Value(t')) \\
throw(Err) & \text{otherwise}
\end{cases}
\end{array}\right.$$

$$
\begin{array}{l}
\mathit{msel}(\mathit{mm}, \mathit{MTArray}(\mathit{al}, t), \mathit{loc}, x\#y\#\mathit{ys}, e_p, \mathit{env}, \mathit{cd}) \overset{\text{def}}{=} \\
\qquad\qquad \textbf{do} \left[\begin{array}{l}
\mathit{kv} \leftarrow \mathit{expr}(x, e_p, \mathit{env}, \mathit{cd}) \\
\lfloor\!=\!\rfloor \begin{cases} \boxed{1} & \text{if } \mathit{kv} = (\mathit{Simple}(v), \mathit{Value}(t')) \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases}
\end{array} \right. \\
\qquad \text{where } \boxed{1} = \mathit{assert}\ \mathit{Err} \left[\begin{array}{l}
\lambda\_.\ \neg \mathit{less}(t', \mathit{TUInt}(256), v, \lfloor \mathit{al} \rfloor) = (\lfloor \mathit{True} \rfloor, \mathit{TBool})_\perp \\
\textbf{do} \left[\begin{array}{l}
s \leftarrow \mathit{apply} \left(\lambda \mathit{st}.\ \begin{cases} \mathit{Memory}(\mathit{st}) & \text{if } \mathit{mm} \\ \mathit{cd} & \text{otherwise} \end{cases} \right) \\
\lfloor\!=\!\rfloor \begin{cases} \mathit{msel}(\mathit{mm}, t, l, (y\#\mathit{ys}), e_p, \mathit{env}, \mathit{cd}) \\ \qquad\qquad \text{if } \mathit{accessStore}(h(\mathit{loc}, v), s) = \mathit{Pointer}(l)_\perp \\ \mathit{throw}(\mathit{Err}) \\ \qquad\qquad \text{otherwise} \end{cases}
\end{array} \right.
\end{array} \right.
\end{array}
$$

$$\mathit{msel}(\_,\_,\_,\_,\_,\_,\_) \overset{\text{def}}{=} \mathit{throw}(\mathit{Err})$$

Note that we require the list of indices to be non-empty. Then, we distinguish two cases. $\mathit{msel}(\_, \mathit{MTArray}(\mathit{al}, t), \mathit{loc}, [x], e_p, \mathit{env}, \mathit{cd})$ captures the case in which the list contains only a single index x. If x evaluates to a basic valuetype v, we check whether v falls within the bounds of the array. If it does, we return the result of hashing the concatenation of the location *loc* with v.

$\mathit{msel}(\mathit{mm}, \mathit{MTArray}(\mathit{al}, t), \mathit{loc}, x\#y\#\mathit{ys}, e_p, \mathit{env}, \mathit{cd})$, on the other hand, captures the case in which the list contains at least two indices. If x evaluates to a valuetype ($\boxed{1}$), we first check that it is within the array's bounds and we then retrieve the pointer stored at location l and pass it to the recursive call of *msel*.

Note that indices are evaluated using the expression function *expr* which may have side effects. Thus, using *msel* to traverse a reference type may also lead to a change in state.

To give an example, let us assume that $t = \mathit{MTArray}(5, \mathit{MTArray}(6, \mathit{MTValue}(\mathit{TBool})))$, and the memory of state s is $[\text{“}\mathit{3.2}\text{”} \mapsto \mathit{Pointer}(\text{“}\mathit{5}\text{”})]$. Then,

$$
\begin{array}{l}
\mathit{msel}(\mathit{True}, t, \text{“}\mathit{2}\text{”}, [\mathit{UInt}(8,3)], \_,\_,\_, s) = \mathit{Normal}\,(\text{“}\mathit{3.2}\text{”}, \mathit{MTArray}(6, \mathit{MTValue}(\mathit{TBool})), s) \\
\mathit{msel}(\mathit{True}, t, \text{“}\mathit{2}\text{”}, [\mathit{UInt}(8,3), \mathit{UInt}(8,4)], \_,\_,\_, s) = \mathit{Normal}\,(\text{“}4.5\text{”}, \mathit{MTValue}(\mathit{TBool}), s) \\
\mathit{msel}(\mathit{True}, t, \text{“}\mathit{2}\text{”}, [\mathit{UInt}(8,5)], \_,\_,\_, s) = \mathit{Exception}(\mathit{Err})
\end{array}
$$

A similar function

$$
\begin{array}{l}
\mathit{ssel}\colon \mathbf{STypes} \times \mathbf{Loc} \times [E] \times \mathbf{Environment}_p \times \mathbf{Environment} \times \mathbf{Calldata} \\
\qquad\qquad\qquad \to \mathbf{State\_Monad}(\mathbf{Loc} \times \mathbf{STypes}, \mathbf{Exception}, \mathbf{State}) \qquad (\texttt{ssel})
\end{array}
$$

is defined to look up storage values:

$$
\begin{array}{l}
\mathit{ssel}(\mathit{STArray}(\mathit{al}, t), \mathit{loc}, x\#\mathit{xs}, e_p, \mathit{env}, \mathit{cd}) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l}
\mathit{kv} \leftarrow \mathit{expr}(x, e_p, \mathit{env}, \mathit{cd}) \\
\lfloor\!=\!\rfloor \begin{cases} \boxed{1} & \text{if } \mathit{kv} = (\mathit{Simple}(v), \mathit{Value}(t')) \\ \mathit{throw}(\mathit{Err}) \\ & \text{otherwise} \end{cases}
\end{array} \right. \\
\qquad \text{where } \boxed{1} = \mathit{assert}\ \mathit{Err} \left[\begin{array}{l}
\lambda\_.\ \neg \mathit{less}(t', \mathit{TUInt}(256), v, \lfloor \mathit{al} \rfloor) = (\lfloor \mathit{True} \rfloor, \mathit{TBool})_\perp \\
\mathit{ssel}(t, h(\mathit{loc}, v), \mathit{xs}, e_p, \mathit{env}, \mathit{cd})
\end{array} \right.
\end{array}
$$

$$ssel(STMap(\_, t), loc, x\#xs, e_p, env, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} kv \leftarrow expr(x, e_p, env, cd) \\ \llcorner\!\!=\!\!\lrcorner \begin{cases} ssel(t, h(loc, v), xs, e_p, env, cd) \\ \qquad\qquad \text{if } kv = (Simple(v), \_) \\ throw(Err) \\ \qquad\qquad \text{otherwise} \end{cases} \end{array} \right.$$

$$ssel(tp, loc, [], \_, \_, \_) \overset{\text{def}}{=} return(loc, tp)$$

$$ssel(\_, \_, \_, \_, \_, \_, \_) \overset{\text{def}}{=} throw(Err)$$

Again, we assume the list of indices to be non-empty. Then, $ssel(STArray(al, t), loc, x\#xs, e_p, env, cd)$ captures the case in which we want to traverse a storage array. If the index evaluates to a valuetype v ($\boxed{1}$), we check if it is within the bounds of the array, in which case we hash the concatenation of location loc with v and recursively call $ssel$ again on the resulting location.

The case for mappings is similar and handled by $ssel(STMap(\_, t), loc, x\#xs, e_p, env, cd)$. The main difference is that we do not need to check that the index is within some bounds since mappings do not have them.

Note that since storage does not support pointer structures, we do not access the store while iterating through the list of selectors.

5.2 Lookup functions

We can now define functions to look up the value of a variable. Note that the meaning of a variable is different depending on whether it is on the left or right-hand side of an assignment.

5.2.1 Left-hand side. We first discuss the meaning of a variable on the left-hand side of an assignment. We first introduce the following type to encapsulate the return value of the corresponding lookup:

$$\textbf{LType} \quad ::= \quad LStackloc(\textbf{Loc}) \mid LMemloc(\textbf{Loc}) \mid LStoreloc(\textbf{Loc}) \qquad (\texttt{LType})$$

The idea is that a variable on the left-hand side always refers to either a location on the stack, memory, or storage. Note that we do not allow for locations in calldata since calldata is read-only, and we are not allowed to assign to it anyway.

We can define the following function to look up the location of a variable on the left-hand side:

$$\begin{aligned} lexp\colon\ & \textbf{L} \times \textbf{Environment}_p \times \textbf{Environment} \times \textbf{Calldata} \\ & \qquad \rightarrow \textbf{State\_Monad}(\textbf{LType} \times \textbf{Type}, \textbf{Exception}, \textbf{State}) \qquad (\texttt{lexp}) \end{aligned}$$

The definition of the function distinguishes two cases depending on whether the variable is dereferenced or not. The latter case is defined as follows:

$$lexp(Id(i), \_, e, \_) \overset{\text{def}}{=} \begin{cases} return(LStackloc(l), tp) & \text{if } denvalue(e)(i) = (tp, Stackloc(l))_\perp \\ return(LStoreloc(l), tp) & \text{if } denvalue(e)(i) = (tp, Storeloc(l))_\perp \\ throw(Err) & \text{otherwise} \end{cases}$$

In this case, the variable refers to either a location on the stack or storage. In either case we just look up the location from the environment and return it as a corresponding **LType**

The case in which the variable is dereferenced using *Ref* is as follows:

$$
\mathit{lexp}(\mathit{Ref}(i,r), e_p, e, cd) \overset{\text{def}}{=}
\begin{cases}
\textbf{do} \left[\begin{array}{l}
k \leftarrow \mathit{apply}(\lambda st.\ \mathit{accessStore}(l, \mathit{sck}(st))) \\
\llcorner\!=\!\lrcorner \begin{cases}
\textbf{do} \left[\begin{array}{l} (l'', t') \leftarrow \mathit{msel}(\mathit{True}, t, l', r, e_p, e, cd) \\ \mathit{return}(\mathit{LMemloc}(l''), \mathit{Memory}(t')) \end{array} \right. \\
\qquad \text{if } k = \mathit{Memptr}(l')_\perp \wedge tp = \mathit{Memory}(t) \\
\textbf{do} \left[\begin{array}{l} (l'', t') \leftarrow \mathit{ssel}(t, l', r, e_p, e, cd) \\ \mathit{return}(\mathit{LStoreloc}(l''), \mathit{Storage}(t')) \end{array} \right. \\
\qquad \text{if } k = \mathit{Stoptr}(l')_\perp \wedge tp = \mathit{Storage}(t) \\
\mathit{throw}(\mathit{Err}) \\
\qquad \text{otherwise}
\end{cases}
\end{array} \right. \\
\qquad \text{if } \mathit{denvalue}(e)(i) = (tp, \mathit{Stackloc}(l))_\perp \\
\textbf{do} \left[\begin{array}{l} (l', t') \leftarrow \mathit{ssel}(t, l, r, e_p, e, cd) \\ \mathit{return}(\mathit{LStoreloc}(l'), \mathit{Storage}(t')) \end{array} \right. \\
\qquad \text{if } \mathit{denvalue}(e)(i) = (tp, \mathit{Storeloc}(l))_\perp \wedge tp = \mathit{Storage}(t) \\
\mathit{throw}(\mathit{Err}) \\
\qquad \text{otherwise}
\end{cases}
$$

Again, the variable either refers to a location on stack or storage. If it refers to a location on stack it can be either a pointer to memory or storage. In the former case we use the *msel* function introduced above to look up the corresponding location in memory. In the latter case we use the *ssel* function instead. For the case the variable refers to a location on storage we can just use the *ssel* function to directly look up the corresponding storage location.

5.2.2 *Right-hand side.* The meaning of a variable occurring on the right-hand side of an assignment is defined by the following function:

$$
\mathit{rexp}\colon \mathbf{L} \times \mathbf{Environment_P} \times \mathbf{Environment} \times \mathbf{Calldata} \longrightarrow \mathbf{State\_Monad}(\mathbf{Stackvalue} \times \mathbf{Type}, \mathbf{Exception}, \mathbf{State}) \qquad (\texttt{rexp})
$$

Again, the definition distinguishes two cases. The case in which the variable is referred to using *Id* is as follows:

$$
\mathit{rexp}(\mathit{Id}(i), e_p, e, cd) \overset{\text{def}}{=}
\begin{cases}
\boxed{1} & \text{if } \mathit{denvalue}(e)(i) = (tp, \mathit{Stackloc}(l))_\perp \\
\boxed{2} & \text{if } \mathit{denvalue}(e)(i) = (\mathit{Storage}(\mathit{STValue}(t)), \mathit{Storeloc}(l))_\perp \\
\boxed{3} & \text{if } \mathit{denvalue}(e)(i) = (\mathit{Storage}(\mathit{STArray}(x,t)), \mathit{Storeloc}(l))_\perp \\
\mathit{throw}(\mathit{Err}) & \text{otherwise}
\end{cases}
$$

$$
\boxed{1} = \textbf{do} \left[\begin{array}{l}
s \leftarrow \mathit{apply}(\lambda st.\ \mathit{accessStore}(l, \mathit{sck}(st))) \\
\llcorner\!=\!\lrcorner \begin{cases}
\mathit{return}(\mathit{Simple}(v), tp) & \text{if } s = \mathit{Simple}(v)_\perp \\
\mathit{return}(\mathit{CDptr}(p), tp) & \text{if } s = \mathit{CDptr}(p)_\perp \\
\mathit{return}(\mathit{Memptr}(p), tp) & \text{if } s = \mathit{Memptr}(p)_\perp \\
\mathit{return}(\mathit{Stoptr}(p), tp) & \text{if } s = \mathit{Stoptr}(p)_\perp \\
\mathit{throw}(\mathit{Err}) & \text{otherwise}
\end{cases}
\end{array} \right.
$$

$$
\boxed{2} = \mathit{option}\ \mathit{Err} \left[\begin{array}{l}
\lambda st.\ \mathit{sto}(st)(\mathit{address}(e)) \\
\lambda s.\ \mathit{return}(\mathit{Simple}(\mathit{accessStorage}(t, l, s)), \mathit{Value}(t))
\end{array} \right.
$$

$$\boxed{3} = \mathit{return}(\mathit{Stoptr}(l, \mathit{Storage}(\mathit{STArray}(x, t))))$$

In the case the variable refers to a location on stack ($\boxed{1}$) we simply look up the location on the stack and return the corresponding value. In the case the variable refers to storage, it is either a valuetype ($\boxed{2}$) or a pointer to an array ($\boxed{3}$) (since pointers to mappings are not allowed). In the former case we again look up the corresponding value from storage and return it. In the latter case we create a corresponding storage pointer which we then return. Note the use of *accessStorage* instead of *accessStore* which ensures that access to storage returns a default value for the case the variable is not initialised.

The case in which the variable is dereferenced is defined as follows:

$$\mathit{rexp}(\mathit{Ref}(i, r), e_p, e, cd) \overset{\text{def}}{=} \begin{cases} \boxed{1} & \text{if } \mathit{denvalue}(e)(i) = (tp, \mathit{Stackloc}(l))_\perp \\ \boxed{2} & \text{if } \mathit{denvalue}(e)(i) = (tp, \mathit{Storeloc}(l))_\perp \land tp = \mathit{Storage}(t) \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases}$$

In particular, we distinguish two cases depending on whether the variable refers to a location on the stack ($\boxed{1}$) or ($\boxed{2}$) storage.

The case in which the variable refers to the stack is as follows:

$$\boxed{1} = \textbf{do} \left[\begin{array}{l} kv \leftarrow \mathit{apply}(\lambda st.\ \mathit{accessStore}(l, \mathit{sck}(st))) \\ \llcorner\!\lrcorner \begin{cases} \boxed{1.1} & \text{if } kv = \mathit{CDptr}(l')_\perp \land tp = \mathit{Calldata}(t) \\ \boxed{1.2} & \text{if } kv = \mathit{Memptr}(l')_\perp \land tp = \mathit{Memory}(t) \\ \boxed{1.3} & \text{if } kv = \mathit{Stoptr}(l')_\perp \land tp = \mathit{Storage}(t) \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{1.1} = \textbf{do} \left[\begin{array}{l} (l'', t') \leftarrow \mathit{msel}(\mathit{False}, t, l', r, e_p, e, cd) \\ \llcorner\!\lrcorner \begin{cases} (\mathit{Simple}(v), \mathit{Value}(t'')) \\ \qquad \text{if } t' = \mathit{MTValue}(t'') \land \mathit{accessStore}(l'', cd) = \mathit{Value}(v)_\perp \\ (\mathit{CDptr}(p), \mathit{Calldata}(\mathit{MTArray}(x, t''))) \\ \qquad \text{if } t' = \mathit{MTArray}(x, t'') \land \mathit{accessStore}(l'', cd) = \mathit{Pointer}(p)_\perp \\ \mathit{throw}(\mathit{Err}) \\ \qquad \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{1.2} = \textbf{do} \left[\begin{array}{l} (l'', t') \leftarrow \mathit{msel}(\mathit{True}, t, l', r, e_p, e, cd) \\ \llcorner\!\lrcorner \begin{cases} \boxed{1.2.1} & \text{if } t' = \mathit{MTValue}(t'') \\ \boxed{1.2.2} & \text{if } t' = \mathit{MTArray}(x, t'') \end{cases} \end{array} \right.$$

$$\boxed{1.2.1} = \textbf{do} \left[\begin{array}{l} mv \leftarrow \mathit{apply}(\lambda st.\ \mathit{accessStore}(l'', \mathit{mem}(st))) \\ \llcorner\!\lrcorner \begin{cases} \mathit{return}(\mathit{Simple}(v), \mathit{Value}(t'')) & \text{if } mv = \mathit{Value}(v)_\perp \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{1.2.2} = \textbf{do} \left[\begin{array}{l} mv \leftarrow \mathit{apply}(\lambda st.\ \mathit{accessStore}(l'', \mathit{mem}(st))) \\ \llcorner\!\lrcorner \begin{cases} \mathit{return}(\mathit{Memptr}(p), \mathit{Memory}(\mathit{MTArray}(x, t''))) & \text{if } mv = \mathit{Pointer}(p)_\perp \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{1.3} = \textbf{do} \left[\begin{array}{l} (l'', t') \leftarrow ssel(t, l', r, e_p, e, cd) \\ \llcorner\!\lrcorner \begin{cases} \boxed{1.3.1} & \text{if } t' = STValue(t'') \\ return(Stoptr(l''), Storage(t')) & \text{if } t' = STArray(\_, \_) \\ return(Stoptr(l''), Storage(t')) & \text{if } t' = STMap(\_, \_) \end{cases} \end{array} \right.$$

$$\boxed{1.3.1} = \textbf{Err} \left[\begin{array}{l} \lambda st.\ sto(st)(address(e)) \\ \lambda s.\ return(Simple(accessStorage(t'', l'', s)), Value(t'')) \end{array} \right.$$

We first look up the corresponding pointer from the stack and distinguish three cases. If the pointer points to calldata ($\boxed{1.1}$), we use function *msel* from above to lookup the corresponding location referred to by the index. Now depending on whether the location refers to a valuetype or to another array we return either a valuetype or a pointer to calldata. The case in which the pointer points to memory ($\boxed{1.2}$) is similar to the calldata case with two notable differences: First, we need to change the first parameter when using *msel* to make sure we use memory instead of calldata. Second, since memory is part of the state we need to first look it up using *apply*. The case in which the pointer points to storage ($\boxed{1.3}$) is again similar except that we now also need to handle the case in which the variable refers to a map. This situation, however, is handled similar to the array case, and we can just return a storage pointer.

The case in which the variable refers to storage directly is just similar to the case in which it refers to a storage pointer in stack:

$$\boxed{2} = \textbf{do} \left[\begin{array}{l} (l', t') \leftarrow ssel(t, l, r, e_p, e, cd) \\ \llcorner\!\lrcorner \begin{cases} \boxed{2.1} & \text{if } t' = STValue(t'') \\ return(Stoptr(l'), Storage(t')) & \text{if } t' = STArray(\_, \_) \\ return(Stoptr(l'), Storage(t')) & \text{if } t' = STMap(\_, \_) \end{cases} \end{array} \right.$$

$$\boxed{2.1} = option\ Err \left[\begin{array}{l} \lambda st.\ sto(st)(address(e)) \\ \lambda s.\ return(Simple(accessStorage(t'', l', s))) \end{array} \right.$$

5.3 Semantics of expressions

We can now define the semantic function for expressions:

$$expr\colon \mathbf{E} \times \mathbf{Environment_P} \times \mathbf{Environment} \times \mathbf{Calldata} \longrightarrow \mathbf{State\_Monad}(\mathbf{Stackvalue} * \mathbf{Type}, \mathbf{Exception}, \mathbf{State}) \qquad (\texttt{expr})$$

The function requires three parameters to compute the semantics of an expression:

- A procedure environment to handle procedure calls
- A variable environment to look up the value of variables
- Calldata to access reference types passed through method calls

It then produces a monad modifying the state and returning a result value and type for the corresponding expression.

As mentioned in Sect. 3.3.2, we assume the existence of a function

$$costs\colon \mathbf{E} \times \mathbf{Environment_P} \times \mathbf{Environment} \times \mathbf{Calldata} \longrightarrow (\mathbf{State} \longrightarrow \mathbf{Nat}) \qquad (\texttt{costs}_\texttt{e})$$

to capture the gas costs associated with executing a certain expression.

In the following we discuss the definition of the function in detail.

5.3.1 Basic expressions. The semantics of the boolean constants is straightforward and defined as follows:

$$expr(True, e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(True, e_p, e, cd)) \\ return(Simple(\lfloor True \rfloor), Value(TBool)) \end{array} \right.$$

$$expr(False, e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(False, e_p, e, cd)) \\ return(Simple(\lfloor False \rfloor), Value(TBool)) \end{array} \right.$$

We only need to ensure that enough Gas is available to execute the command for which we use the *gascheck* monad. Then we can just return a textual representation of the corresponding constant.

The semantics of the expression to create addresses is similar:

$$expr(Address(ad), e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(Address(ad), e_p, e, cd)) \\ return(Simple(ad), Value(TAddr)) \end{array} \right.$$

We simply create a corresponding stackvalue element and type.

The semantics of integer constants is also straightforward and defined as follows:

$$expr(SInt(b, x), e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(SInt(b, x), e_p, e, cd)) \\ return(Simple(createSInt(b, x)), Value(TSInt, b)) \end{array} \right.$$

$$expr(UInt(b, x), e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(UInt(b, x), e_p, e, cd)) \\ return(Simple(createUInt(b, x)), Value(TUInt, b)) \end{array} \right.$$

Note, however, that we use functions *createSInt* and *createUInt* to handle overflow when converting a number to a certain integer representation.

The values of the constants denoting the currently executing contract, the message sender, and the transferred funds can be obtained from the current environment:

$$expr(This, e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(This, e_p, e, cd)) \\ return(Simple(address(e)), Value(TAddr)) \end{array} \right.$$

$$expr(Sender, e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(Sender, e_p, e, cd)) \\ return(Simple(sender(e)), Value(TAddr)) \end{array} \right.$$

$$expr(Value, e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(Value, e_p, e, cd)) \\ return(Simple(svalue(e)), Value(TUInt(256))) \end{array} \right.$$

Note that the amount of funds is returns as a 256-bit unsigned integer.

Having defined function *rexp* above makes it easy to define the semantics of lvalues:

$$expr(L(i), e_p, e, cd) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} gascheck(costs(L(i), e_p, e, cd)) \\ rexp(i, e_p, e, cd) \end{array} \right.$$

We just need to ensure that enough gas is available and can then simply pass on the identifier to *rexp*.

To obtain the balance of an address we first need to evaluate the corresponding expression to determine the actual address:

$$expr(Balance(ad), e_p, e, cd) \overset{\text{def}}{=}$$

$$\textbf{do}\left[\begin{array}{l} \mathit{gascheck}(\mathit{costs}(\mathit{Balance}(ad), e_p, e, cd)) \\ kv \longleftarrow \mathit{expr}(ad, e_p, e, cd) \\ \mathbin{\llcorner\!-\!\lrcorner} \begin{cases} \mathit{return}(\mathit{Simple}(\mathit{acc}(st)(adv)), \mathit{Value}(\mathit{TUInt}(256))) & \\ \qquad\qquad \text{if } kv = (\mathit{Simple}(adv), \mathit{Value}(\mathit{TAddr})) & \\ \mathit{throw}(\mathit{Err}) \qquad\qquad \text{otherwise} & \end{cases} \end{array}\right.$$

For the case the expression does not evaluate to an address we return an exception *Err*. Again, the balance is returned as a 256-bit unsigned integer.

5.3.2 Compound expressions. The semantic of negation is straightforward and defined as follows:

$$\mathit{expr}(\neg x, e_p, e, cd) \overset{\text{def}}{=} \textbf{do}\left[\begin{array}{l} \mathit{gascheck}(\mathit{costs}(\neg x, e_p, e, cd)) \\ kv \longleftarrow \mathit{expr}(x, e_p, e, cd) \\ \mathbin{\llcorner\!-\!\lrcorner} \begin{cases} \mathit{expr}(\mathit{False}, e_p, e, cd) & \text{if } kv = (\mathit{Simple}(\lfloor \mathit{True} \rfloor), \mathit{Value}(\mathit{TBool})) \\ \mathit{expr}(\mathit{True}, e_p, e, cd) & \text{if } kv = (\mathit{Simple}(\lfloor \mathit{False} \rfloor), \mathit{Value}(\mathit{TBool})) \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases} \end{array}\right.$$

We just need to compute the result of the original value x and inspect its string representation.

The semantics of the addition operation is defined as follows:

$$\mathit{expr}(e_1{+}e_2, e_p, e, cd) \overset{\text{def}}{=} \textbf{do}\left[\begin{array}{l} \mathit{gascheck}(\mathit{costs}(e_1{+}e_2, e_p, e, cd)) \\ \mathit{kv1} \longleftarrow \mathit{expr}(e_1, e_p, e, cd) \\ \mathbin{\llcorner\!-\!\lrcorner} \begin{cases} \boxed{1} & \text{if } \mathit{kv1} = (\mathit{Simple}(v_1), \mathit{Value}(t_1)) \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1} = \textbf{do}\left[\begin{array}{l} \mathit{kv2} \longleftarrow \mathit{expr}(e_2, e_p, e, cd) \\ \mathbin{\llcorner\!-\!\lrcorner} \begin{cases} \boxed{1.1} & \text{if } \mathit{kv2} = (\mathit{Simple}(v_2), \mathit{Value}(t_2)) \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1.1} = \mathit{option}\ \mathit{Err}\left[\begin{array}{l} \lambda\_.\ \mathit{add}(t_1, t_2, v_1, v_2) \\ \lambda(v, t).\ \mathit{return}(\mathit{Simple}(v), \mathit{Value}(t)) \end{array}\right.$$

Note that we use the function *add* defined above to deal with potential overflows. In addition, note that evaluating the first operand may have side effects on the state which impact the evaluation of the second operand. The semantics of the remaining arithmetic and logical operators is similar and not discussed further here.

5.3.3 Method calls. To define the semantics of method calls we first need to define a function

$$\begin{aligned} \mathit{load}\colon & \textbf{Bool} \times [\textbf{Identifier} \times \textbf{Type}] \times [\textbf{E}] \times \textbf{Environment}_{\textbf{P}} \times \textbf{Environment} \times \textbf{Calldata} \\ & \times \textbf{Stack} \times \textbf{Memory} \times \textbf{Environment} \times \textbf{Calldata} \\ & \longrightarrow \textbf{State\_Monad}(\textbf{Environment} \times \textbf{Calldata} \times \textbf{Stack} \times \textbf{Memory}, \textbf{Exception}, \textbf{State}) \end{aligned} \qquad (\texttt{load})$$

to load the methods parameters:

$$\mathit{load}(cp, (i_p, t_p)\#pl, e\#el, e_p, e_v', cd', sck', mem', e_v, cd) \overset{\text{def}}{=} \textbf{do}\left[\begin{array}{l} (v, t) \longleftarrow \mathit{expr}(e, e_p, e_v, cd) \\ \boxed{1} \end{array}\right.$$

$$\boxed{1} = \mathit{option}\ \mathit{Err}\left[\begin{array}{l} \lambda st.\ \mathit{decl}(i_p, t_p, (v, t)_\perp, cp, cd, \mathit{mem}(st), \mathit{sto}(st), (cd', mem', sck', e_v')) \\ \lambda(c, m, k, e).\ \mathit{load}(cp, pl, el, e_p, e, c, k, m, e_v, cd) \end{array}\right.$$

$$\mathit{load}(\_, [], [], \_, e_v', cd', sck', mem', e_v, cd) \overset{\text{def}}{=} \mathit{return}(e_v', cd', sck', mem')$$

$$load(\_, \_, \_, \_, \_, \_, \_, \_, \_, \_) \overset{\text{def}}{=} throw(Err)$$

Roughly speaking, the function takes a list of formal parameters and expressions and creates an environment in which the parameters are initialized with the values obtained from the corresponding expression. Note that loading the parameters may have side effects and change the overall state.

In particular, the idea is that

$$load(cp, fp, es, env_p, env, cd_o, k, m, e, c)(st) = Normal((e', c', k', m'), st')$$

creates a new environment with corresponding stack, calldata and memory. Thus, we cannot just update the corresponding elements from the state *st* but rather need to carry separate copies of these elements as parameters e, k, c, and m. The function the updates these elements to e', k', c', and m' by adding variables from *fp* with values obtained from evaluating *es*. When evaluating the expressions, however, we do rely on the original environment, stack, calldata, and memory from the state *st*. In addition, evaluation of the expressions may change the state which is why the function also returns an updated version of the state st'. Parameter *env* and *cd* represent the original versions of the environment and calldata when starting the loading process.

In Solidity, method arguments are evaluated from left to right as demonstrated by the following example.

Example 5.1 (Method calls). Consider the following contract:

```
contract Example {
  function inc1(uint8[1] memory x)
    private returns (uint8) {
    assert (x[0] == 0);
    x[0] = 1;
    return 0;
  }
  function inc2(uint8[1] memory x)
    private returns (uint8) {
    assert (x[0] == 1);
    x[0] = 2;
    return 0;
  }
  function test(uint a, uint8 b)
      private returns (uint8) {
    return 0;
  }
  function test2() public {
    uint8[1] memory x=[0];
    test(inc1(x), inc2(x));
  }




}
```

The contract provides two methods `inc1` and `inc2` which modify the only entry of an integer memory array. However, while `inc1` sets the value to 0, `inc2` sets it to one. Now to test the evaluation of arguments we provide a third function `test` which requires two integers as input. Finally, we can define a method `test2` which just invokes `test` with parameter values given by `inc1` and `inc2`, respectively. Now, since both, `inc1` and `inc2` modify a member variable of the contract we can check the evaluation of arguments using corresponding assertions. In our example we can see that `inc1` is evaluated first as otherwise the assertions would fail.

The semantics of method calls requires looking up the corresponding statement from the procedure environment, prepare the environment, and execute the statement. The semantics of internal calls is defined as follows:

$$expr(Call(i, xe), e_p, e, cd) \overset{\text{def}}{=} \mathbf{do} \left[\begin{array}{l} gascheck(costs(Call(i, xe), e_p, e, cd)) \\ \llcorner\!-\!\lrcorner \begin{cases} \boxed{1} & \text{if } e_p(address(e)) = (ct, \_)_\bot \\ & \quad \wedge ct(i) = Method(fp, f, x_\bot)_\bot \\ throw(Err) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1} = \mathbf{do} \left[\begin{array}{l} m_o \longleftarrow apply(\lambda st.\ mem(st)) \\ (e_l, cd_l, k_l, m_l) \longleftarrow load(False, fp, xe, e_p, e', \\ \qquad\qquad emptyStore, emptyStore, m_o, e, cd) \\ k_o \longleftarrow apply(\lambda st.\ sck(st)) \\ modify(\lambda st.\ upMem(m_l, upSck(k_l, st))) \\ stmt(f, e_p, e_l, cd_l) \\ rv \longleftarrow expr(x, e_p, e_l, cd_l) \\ modify(\lambda st.\ upSck(k_o, st)) \\ return(rv) \end{array}\right.$$

where $e' = \mathit{ffold}(init(ct), empty(address(e), sender(e), svalue(e)), dom(ct))$.

To execute a method i on the current contract, we first need to look up the corresponding method from the procedure environment e_p. Then ($\boxed{1}$), we create a fresh stack and use the *ffold* function defined above to create a fresh variable environment e' for the execution of the method. Next we load the parameters of the method into the new environment to obtain another environment e''. Since reference types used in arguments are stored in calldata, the load command also returns a calldata store cd'. Moreover, evaluating the parameters may have side effects which result in a new state st''. We then keep a copy of the original state in st''' and replace it with the new state st'' obtained from loading the parameters. Finally, we can execute the statement of the method using the new environment e'' and calldata cd'. We then evaluate the methods return expression rv and reset the stack to how it was in state st''' before returning the actual value rv.

The semantics of external method calls is defined as follows:

$$expr(ECall(ad, i, xe, val), e_p, e, cd)$$

$$\overset{\text{def}}{=} \mathbf{do} \left[\begin{array}{l} gascheck(costs(ECall(ad, i, xe, val), e_p, e, cd)) \\ kad \longleftarrow expr(ad, e_p, e, cd) \\ \llcorner\!-\!\lrcorner \begin{cases} \boxed{1} & \text{if } kad = (Simple(adv), Value(TAddr)) \\ & \quad \wedge\ e_p(adv) = (ct, \_)_\bot \wedge ct(i) = Method(fp, f, x_\bot)_\bot \\ throw(Err) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1} = \mathbf{do} \left[\begin{array}{l} kv \longleftarrow expr(val, e_p, e, cd) \\ \llcorner\!-\!\lrcorner \begin{cases} \boxed{1.1} & \text{if } kv = (Simple(v), Value(t)) \\ throw(Err) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1.1} = \mathbf{do} \left[\begin{array}{l} (e_l, cd_l, k_l, m_l) \longleftarrow load(True, fp, xe, e_p, e', emptyStore, emptyStore, emptyStore, e, cd) \\ option\ Err \left[\begin{array}{l} \lambda st.\ transfer(address(e), adv, v, acc(st)) \\ \lambda acc.\ \mathbf{do} \left[\begin{array}{l} (k_o, m_o) \longleftarrow apply(\lambda st.\ (sck(st), mem(st))) \\ modify(\lambda st.\ upMem(m_l, upSck(k_l, upAcc(acc, st)))) \\ stmt(f, e_p, e_l, cd_l) \\ rv \longleftarrow expr(x, e_p, e_l, cd_l) \\ modify(\lambda st.\ upMem(m_o, upSck(k_o, st))) \\ return(rv) \end{array}\right. \end{array}\right. \end{array}\right.$$

Here, we first need to evaluate expression *ad* to obtain the address of the target contract and lookup the corresponding method from the procedure environment e_p. Next ($\boxed{1}$) we need to evaluate expression *val* to obtain the amount of funds to be sent with the call. Then ($\boxed{1.1}$) we prepare a new state st' with a fresh stack and memory as well as a new environment e' using function *ffold* described above. We can then use the new state and environment to load the method parameters into a new environment e'' using function *load* defined above. We then keep a copy of the current state st''' and transfer the funds sent with the method call from the calling contract to the called one. We execute the body of the method on an updated version of state st'' which contains the new balances after transferring the funds. Finally, we compute the return value *rv*, restore the state of the stack and memory, and return *rv*.

Note that we return an error in the case that we call a method identifier which does not exist. In Solidity such a call would actually be allowed and trigger an execution of the fallback function of the contract. However, as soon as we try to access the expected return value we would get an exception which reverts the execution.

Consider, for example, the following two contracts:

```
1  contract Caller {
2    function tCall(address _ad, string memory _mt) public returns (uint) {
3      (bool s, bytes memory d) = _ad.call.gas(5000)(
4        abi.encodeWithSignature(_mt)
5        );
6      return abi.decode(d, (uint));
7    }
8  }
```

```
1  contract Receiver {
2    event Received(string message);
3
4    function () external {
5      emit Received("Fallback called");
6    }
7    function test() public returns (uint) {
8      return 1;
9    }
10 }
```

Executing function `tCall` of contract `Caller` with the address of the `Receiver` contract and method parameter “test()” will execute function `test` of the receiver contract and return a 1. Now, if we misspell the method name and use “ttest()” instead, contract `Receiver` would execute its fallback function instead. However, since the fallback function does not return any value, the decoding in Line 6 of the `Caller` contract would throw an exception and revert the execution and the `Receiver` contracts fallback function is never executed. Thus, our semantics indeed faithfully models the execution of an external method call used in an expression.

Note also that there are several differences between internal and external method calls. In particular, internal method calls keep the original memory while external calls use a fresh memory to execute. Moreover, only external calls allow funds to be transferred with the call. The transfer triggered by an external method call does *not* trigger the fallback function.

6 STATEMENTS

Our subset of Solidity supports the following types of statements:

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

In particular, we support a specific statement to transfer funds as well as internal and external method calls.

The semantics of statements is given by function

$$\begin{aligned}\mathit{stmt}\colon & \mathrm{S} \times \mathbf{Environment_P} \times \mathbf{Environment} \times \mathbf{Calldata} \\ & \rightarrow \mathbf{State\_Monad}((), \mathbf{Exception}, \mathbf{State}) \qquad (\texttt{stmt})\end{aligned}$$

which requires a procedure environment and a calldata store and returns a monad capturing the semantics of a statement. Similarly, as for expressions we assume the existence of a cost function for statements:

$$\mathit{costs}\colon \mathrm{S} \times \mathbf{Environment_P} \times \mathbf{Environment} \times \mathbf{Calldata} \rightarrow (\mathbf{State} \rightarrow \mathbf{Nat}) \qquad (\texttt{costs})$$

Statements are formalized in theory `Statements.thy`. In the following, we discuss the definition of *stmt* in more detail. We first present the semantics of basic statements such as skip and assign (Sect. 6.1). We then discuss the semantics of composition, conditionals, loops, and block statements (Sect. 6.2). Finally, we discuss the semantics of internal (Sect. 6.3) and external (Sect. 6.4) method invocations as well as the transfer statement (Sect. 6.5).

6.1 Basics

The semantics of the *Skip* is straightforward:

$$\mathit{stmt}(\mathit{Skip}, e_p, e, \mathit{cd}) \overset{\text{def}}{=} \mathit{gascheck}(\mathit{costs}(\mathit{Skip}, e_p, e, \mathit{cd}))$$

We just need to ensure that enough gas is available before we return.

The semantics of assignments is a bit special in Solidity and thus worth a closer look:

$$\mathit{stmt}(\mathit{lv}{=}\mathit{ex}, e_p, \mathit{env}, \mathit{cd}) \overset{\text{def}}{=} \mathbf{do} \left[\begin{array}{l} \mathit{gascheck}(\mathit{costs}(\mathit{lv}{=}\mathit{ex}, e_p, \mathit{env}, \mathit{cd})) \\ \mathit{re} \leftarrow \mathit{expr}(\mathit{ex}, e_p, \mathit{env}, \mathit{cd}) \\ \sqcup \begin{cases} \boxed{1} & \text{if } \mathit{re} = (\mathit{Simple}(v), \mathit{Value}(t)) \\ \boxed{2} & \text{if } \mathit{re} = (\mathit{CDptr}(p), \mathit{Calldata}(\mathit{MTArray}(x, t))) \\ \boxed{3} & \text{if } \mathit{re} = (\mathit{Memptr}(p), \mathit{Memory}(\mathit{MTArray}(x, t))) \\ \boxed{4} & \text{if } \mathit{re} = (\mathit{Stoptr}(p), \mathit{Storage}(\mathit{STArray}(x, t))) \\ \boxed{5} & \text{if } \mathit{re} = (\mathit{Stoptr}(p), \mathit{Storage}(\mathit{STMap}(t, t'))) \\ \mathit{throw}(\mathit{Err}) & \\ & \text{otherwise} \end{cases} \end{array}\right.$$

After checking the available gas we first evaluate the right-hand side of the assignment. Depending on the outcome of this we distinguish five different cases.

The first case deals with the situation in which the right-hand side evaluates to a simple valuetype on the stack:

$$\boxed{1} = \mathbf{do} \left[\begin{array}{l} rl \leftarrow lexp(lv, e_p, env, cd) \\ \gg\!= \begin{cases} \boxed{1.1} & \text{if } rl = (LStackloc(l), Value(t')) \\ \boxed{1.2} & \text{if } rl = (LStoreloc(l), Storage(STValue(t'))) \\ \boxed{1.3} & \text{if } rl = (LMemloc(l), Memory(MTValue(t'))) \\ throw(Err) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{1.1} = option\ Err \left[\begin{array}{l} \lambda\_.\ convert(t, t', v) \\ \lambda(v', \_).\ modify(\lambda st.\ upSck(updateStore(l, Simple(v'), sck(st)), st)) \end{array} \right.$$

$$\boxed{1.2} = option\ Err \left[\begin{array}{l} \lambda\_.\ convert(t, t', v) \\ \lambda(v', \_).\ option\ Err \left[\begin{array}{l} \lambda st.\ sto(st)(address(env)) \\ \lambda s.\ modify(\lambda st. \\ \qquad upSto(sto(st)[address(env) \mapsto s[l \mapsto v']], st)) \end{array} \right. \end{array} \right.$$

$$\boxed{1.3} = option\ Err \left[\begin{array}{l} \lambda\_.\ convert(t, t', v) \\ \lambda(v', \_).\ modify(\lambda st.\ upMem(updateStore(l, Value(v'), mem(st)), st)) \end{array} \right.$$

In this case the corresponding value is just copied from the stack to the storage location given by the left-hand side. Note, however, that we need to use function *convert* to convert between the types of the value on the stack and the target destination.

The next case deals with the situation in which the right-hand side is a value kept in calldata:

$$\boxed{2} = \mathbf{do} \left[\begin{array}{l} rl \leftarrow lexp(lv, e_p, env, cd) \\ \gg\!= \begin{cases} \boxed{2.1} & \text{if } rl = (LStackloc(l), Memory(\_)) \\ \boxed{2.2} & \text{if } rl = (LStackloc(l), Storage(\_)) \\ \boxed{2.3} & \text{if } rl = (LStoreloc(l), \_) \\ \boxed{2.4} & \text{if } rl = (LMemloc(l), \_) \\ throw(Err) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{2.1} = \mathbf{do} \left[\begin{array}{l} sv \leftarrow apply(\lambda st.\ accessStore(l, sck(st))) \\ \gg\!= \begin{cases} \boxed{2.1.1} & \text{if } sv = Memptr(p')_{\perp} \\ throw(Err) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{2.1.1} = option\ Err \left[\begin{array}{l} \lambda st.\ cp^m_m(p, p', x, t, cd, mem(st)) \\ \lambda m.\ modify(\lambda st.\ upSto(m, st)) \end{array} \right.$$

$$\boxed{2.2} = \mathbf{do} \left[\begin{array}{l} sv \leftarrow apply(\lambda st.\ accessStore(l, sck(st))) \\ \gg\!= \begin{cases} \boxed{2.2.1} & \text{if } sv = Stoptr(p')_{\perp} \\ throw(Err) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{2.2.1} = option\ Err \left[\begin{array}{l} \lambda st.\ sto(st)(address(env)) \\ \lambda s.\ option\ Err \left[\begin{array}{l} \lambda\_.\ cp^m_s(p, p', x, t, cd, s) \\ \lambda s'.\ modify(\lambda st.\ upSto(sto(st)[address(env) \mapsto s'], st)) \end{array} \right. \end{array} \right.$$

$$\boxed{2.3} = option\ Err \left[\begin{array}{l} \lambda st.\ sto(st)(address(env)) \\ \lambda s.\ option\ Err \left[\begin{array}{l} \lambda\_.\ cp^m_s(p, l, x, t, cd, s) \\ \lambda s'.\ modify(\lambda st.\ upSto(sto(st)[address(env) \mapsto s'], st)) \end{array} \right. \end{array} \right.$$

$$\boxed{2.4} = \mathit{option\ Err} \left[\begin{array}{l} \lambda st.\ cp_m^m(p, l, x, t, cd, mem(st)) \\ modify(\lambda st.\ upMem(m, st)) \end{array} \right.$$

Again, we first evaluate the left-hand side of the assignment to determine the corresponding type of store and location. The left-hand side may be either a pointer to memory ($\boxed{2.1}$, $\boxed{2.4}$) or storage ($\boxed{2.2}$, $\boxed{2.3}$)[5]. In either case we need to copy the structure from calldata to either memory or storage (using the functions cp_m^m or cp_m^s, respectively) and update the corresponding store in the state.

The case in which the right-hand side is a pointer to memory is defined as follows:

$$\boxed{3} = \mathbf{do} \left[\begin{array}{l} rl \longleftarrow lexp(lv, e_p, env, cd) \\ \llcorner\!\lrcorner \begin{cases} modify(\lambda st.\ upSck(updateStore(l, Memptr(p), sck(st)), st)) & \\ \qquad\qquad \text{if } rl = (LStackloc(l), Memory(\_)) & \\ \boxed{3.1} \qquad\quad \text{if } rl = (LStackloc(l), Storage(\_)) & \\ \boxed{3.2} \qquad\quad \text{if } rl = (LStoreloc(l), \_) & \\ modify(\lambda st.\ upMem(updateStore(l, Pointer(p), mem(st)), st)) & \\ \qquad\qquad \text{if } rl = (LMemloc(l), \_) & \\ throw(Err) & \\ \qquad\qquad \text{otherwise} & \end{cases} \end{array} \right.$$

$$\boxed{3.1} = \mathbf{do} \left[\begin{array}{l} sv \longleftarrow apply(\lambda st.\ accessStore(l, sck(st))) \\ \llcorner\!\lrcorner \begin{cases} \boxed{3.1.1} & \text{if } sv = Stoptr(p')_\perp \\ throw(Err) & \text{otherwise} \end{cases} \end{array} \right.$$

$$\boxed{3.1.1} = \mathit{option\ Err} \left[\begin{array}{l} \lambda st.\ sto(st)(address(env)) \\ \lambda s.\ \mathit{option\ Err} \left[\begin{array}{l} \lambda st.\ cp_s^m(p, p', x, t, mem(st), s) \\ \lambda s'.\ modify(\lambda st.\ upSto(sto(st)[address(env) \mapsto s'], st)) \end{array} \right. \end{array} \right.$$

$$\boxed{3.2} = \mathit{option\ Err} \left[\begin{array}{l} \lambda st.\ sto(st)(address(env)) \\ \lambda s.\ \mathit{option\ Err} \left[\begin{array}{l} \lambda st.\ cp_s^m(p, l, x, t, mem(st), s) \\ \lambda s'.\ modify(\lambda st.\ upSto(sto(st)[address(env) \mapsto s'], st)) \end{array} \right. \end{array} \right.$$

As usual we first evaluate the left-hand side of the assignment to obtain a store and location. The semantics of the assignment now depends on the type of store referred to by the left-hand side. For the case the left-hand side points to a location in memory, we just update the pointer referred to by the right-hand side. For the case the left-hand side points to a storage location ($\boxed{3.1}$, $\boxed{3.2}$) the situation is different. In such a case the semantics of an assignment requires us to copy the whole structure from memory to the corresponding storage location referred to by the left-hand side.

[5]Note that Solidity v.5.16 does not support pointers to calldata which is why we do not need to consider this case here.

The next case to consider is the case in which the right-hand side is a reference to a storage array:

$$
\boxed{4} = \textbf{do}\left[\begin{array}{l}
rl \longleftarrow lexp(lv, e_p, env, cd) \\
\sqcup \begin{cases}
\boxed{4.1} & \text{if } rl = (LStackloc(l), Memory(\_)) \\
modify(\lambda st.\ upSck(updateStore(l, Stoptr(p), sck(st)), st)) & \\
 & \text{if } rl = (LStackloc(l), Storage(\_)) \\
\boxed{4.2} & \text{if } rl = (LStoreloc(l), \_) \\
\boxed{4.3} & \text{if } rl = (LMemloc(l), \_) \\
throw(Err) & \\
 & \text{otherwise}
\end{cases}
\end{array}\right.
$$

$$
\boxed{4.1} = \textbf{do}\left[\begin{array}{l}
sv \longleftarrow apply(\lambda st.\ accessStore(l, sck(st))) \\
\sqcup \begin{cases}
\boxed{4.1.1} & \text{if } sv = Memptr(p')_\perp \\
throw(Err) & \text{otherwise}
\end{cases}
\end{array}\right.
$$

$$
\boxed{4.1.1} = option\ Err\left[\begin{array}{l}
\lambda st.\ sto(st)(address(env)) \\
\lambda s.\ option\ Err\left[\begin{array}{l}
\lambda st.\ cp^s_m(p, p', x, t, s, mem(st)) \\
\lambda m.\ modify(\lambda st.\ upMem(m, st))
\end{array}\right.
\end{array}\right.
$$

$$
\boxed{4.2} = option\ Err\left[\begin{array}{l}
\lambda st.\ sto(st)(address(env)) \\
\lambda s.\ option\ Err\left[\begin{array}{l}
\lambda\_.\ cp^s_s(p, l, x, t, s) \\
\lambda s'.\ modify(\lambda st.\ upSto(sto(st)[address(env) \mapsto s'], st))
\end{array}\right.
\end{array}\right.
$$

$$
\boxed{4.3} = option\ Err\left[\begin{array}{l}
\lambda st.\ sto(st)(address(env)) \\
\lambda s.\ option\ Err\left[\begin{array}{l}
\lambda st.\ cp^s_m(p, l, x, t, s, mem(st)) \\
\lambda m.\ modify(\lambda st.\ upMem(m, st))
\end{array}\right.
\end{array}\right.
$$

Again, we first evaluate the left-hand side to obtain the store and location and proceed as follows: In the case the left-hand side evaluates to a memory location ($\boxed{4.1}$, $\boxed{4.3}$) we need to copy the structure from memory to storage. In the case the left-hand side evaluates to a storage pointer, we can just update the pointer. In the case the left-hand side evaluates to a storage location ($\boxed{4.2}$), we need to copy the structure from the location referred to be the right-hand side to the location referred to by the left-hand side.

The final case to consider is when the right-hand side is a reference to a map:

$$
\boxed{5} = \textbf{do}\left[\begin{array}{l}
rl \longleftarrow lexp(lv, e_p, env, cd) \\
\sqcup \begin{cases}
modify(\lambda st.\ upSck(updateStore(l, Stoptr(p), sck(st)), st)) & \\
 & \text{if } rl = (LStackloc(l), \_) \\
throw(Err) & \\
 & \text{otherwise}
\end{cases}
\end{array}\right.
$$

Since maps cannot be copied we do only need to consider the case in which the left-hand side is a storage pointer in which case we just update the pointer to the location of the map.

6.2 Composed statements

Composition of two statements is defined according to the traditional definition of composition for denotational semantics:

$$
\mathit{stmt}(s_1 \mathbin{;} s_2, e_p, e, \mathit{cd}) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} \mathit{gascheck}(\mathit{costs}(s_1 \mathbin{;} s_2, e_p, e, \mathit{cd})) \\ \mathit{stmt}(s_1, e_p, e, \mathit{cd}) \\ \mathit{stmt}(s_2, e_p, e, \mathit{cd}) \end{array} \right.
$$

Similarly, the semantics of the conditional statement is defined as expected if we consider that value types are represented as strings:

$$
\mathit{stmt}(\mathit{Ite}(\mathit{ex}, s_1, s_2), e_p, e, \mathit{cd}) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} \mathit{gascheck}(\mathit{costs}(\mathit{Ite}(\mathit{ex}, s_1, s_2), e_p, e, \mathit{cd})) \\ v \leftarrow \mathit{expr}(\mathit{ex}, e_p, e, \mathit{cd}) \\ \llcorner\!-\!\lrcorner \begin{cases} \mathit{stmt}(s_1, e_p, e, \mathit{cd}) & \text{if } v = (\mathit{Simple}(b), \mathit{Value}(\mathit{TBool})) \\ \qquad \wedge b = \lfloor \mathit{True} \rfloor & \\ \mathit{stmt}(s_2, e_p, e, \mathit{cd}) & \text{if } v = (\mathit{Simple}(b), \mathit{Value}(\mathit{TBool})) \\ \qquad \wedge b = \lfloor \mathit{False} \rfloor & \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases} \end{array} \right.
$$

Similarly, the definition of the loop statement is standard denotational semantics:

$$
\mathit{stmt}(\mathit{While}(\mathit{ex}, s), e_p, e, \mathit{cd}) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} \mathit{gascheck}(\mathit{costs}(\mathit{While}(\mathit{ex}, s), e_p, e, \mathit{cd})) \\ v \leftarrow \mathit{expr}(\mathit{ex}, e_p, e, \mathit{cd}) \\ \llcorner\!-\!\lrcorner \begin{cases} \boxed{1} & \text{if } v = (\mathit{Simple}(b), \mathit{Value}(\mathit{TBool})) \\ & \qquad \wedge b = \lfloor \mathit{True} \rfloor \\ \mathit{return}(()) & \text{if } v = (\mathit{Simple}(b), \mathit{Value}(\mathit{TBool})) \\ & \qquad \wedge b = \lfloor \mathit{False} \rfloor \\ \mathit{throw}(\mathit{Err}) & \text{otherwise} \end{cases} \end{array} \right.
$$

$$
\boxed{1} = \textbf{do} \left[\begin{array}{l} \mathit{stmt}(s, e_p, e, \mathit{cd}) \\ \mathit{stmt}(\mathit{While}(\mathit{ex}, s), e_p, e, \mathit{cd}) \end{array} \right.
$$

The semantics of a block statement is defined using the *decl* function introduced above:

$$
\mathit{stmt}(\mathit{Block}((\mathit{id}, \mathit{tp}, \mathit{ex}), s), e_p, e_v, \mathit{cd}) \overset{\text{def}}{=} \textbf{do} \left[\begin{array}{l} \mathit{gascheck}(\mathit{costs}(\mathit{Block}((\mathit{id}, \mathit{tp}, \mathit{ex}), s), e_p, e, \mathit{cd})) \\ \llcorner\!-\!\lrcorner \begin{cases} \boxed{1} & \text{if } \mathit{ex} = \bot \\ \boxed{2} & \text{if } \mathit{ex} = \mathit{ex}'_{\bot} \end{cases} \end{array} \right.
$$

$$
\boxed{1} = \mathit{option}\ \mathit{Err} \left[\begin{array}{l} \lambda \mathit{st}.\ \mathit{decl}(\mathit{id}, \mathit{tp}, \bot, \mathit{False}, \mathit{cd}, \mathit{mem}(\mathit{st}), \mathit{sto}(\mathit{st}), (\mathit{cd}, \mathit{mem}(\mathit{st}), \mathit{sck}(\mathit{st}), e_v)) \\ \lambda(\mathit{cd}', \mathit{mem}', \mathit{sck}', e').\ \textbf{do} \left[\begin{array}{l} \mathit{modify}(\lambda \mathit{st}.\ \mathit{upMem}(\mathit{mem}', \mathit{upSck}(\mathit{sck}', \mathit{st}))) \\ \mathit{stmt}(s, e_p, e', \mathit{cd}') \end{array} \right. \end{array} \right.
$$

$\boxed{2}$ =

$$
\textbf{do} \left[\begin{array}{l} (v, t) \leftarrow \mathit{expr}(\mathit{ex}', e_p, e_v, \mathit{cd}) \\ \mathit{option}\ \mathit{Err} \left[\begin{array}{l} \lambda \mathit{st}.\ \mathit{decl}(\mathit{id}, \mathit{tp}, (v, t)_{\bot}, \mathit{False}, \mathit{cd}, \mathit{mem}(\mathit{st}), \mathit{sto}(\mathit{st}), (\mathit{cd}, \mathit{mem}(\mathit{st}), \mathit{sck}(\mathit{st}), e_v)) \\ \lambda(\mathit{cd}', \mathit{mem}', \mathit{sck}', e').\ \textbf{do} \left[\begin{array}{l} \mathit{modify}(\lambda \mathit{st}.\ \mathit{upMem}(\mathit{mem}', \mathit{upSck}(\mathit{sck}', \mathit{st}))) \\ \mathit{stmt}(s, e_p, e', \mathit{cd}') \end{array} \right. \end{array} \right. \end{array} \right.
$$

In particular we distinguish two cases: The first case ($\boxed{1}$) deals with situations in which a variable is declared but not initialized. In such a case the variable should be initialized with its default value which is done by function *decl* by passing it $\bot$ as 3rd parameter. The second case ($\boxed{2}$) deals with situations in which a variable is declared and initialized in which case we just forward the initialized value to *decl*. Note that local declarations do not copy structures between memory which is why the 4th parameter of *decl* is *False* in both cases.

6.3 Method invocations

Invocations of internal methods are defined similar to internal method calls in expressions:

$$stmt(Invoke(i, xe), e_p, e, cd) \overset{\text{def}}{=}$$

$$\textbf{do} \left[\begin{array}{l} gascheck(costs(Invoke(i, xe), e_p, e, cd)) \\ \llcorner\!-\!\lrcorner \begin{cases} \boxed{1} & \text{if } e_p(address(e)) = (ct, \_)_\bot \wedge ct(i) = Method(fp, f, \bot)_\bot \\ throw(Err) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1} = \textbf{do} \left[\begin{array}{l} m_o \leftarrow apply(\lambda st.\ mem(st)) \\ (e_l, cd_l, k_l, m_l) \leftarrow load(False, fp, xe, e_p, e', emptyStore, emptyStore, m_o, e, cd) \\ k_o \leftarrow apply(\lambda st.\ sck(st)) \\ modify(\lambda st.\ upMem(m_l, upSck(k_l, st))) \\ stmt(f, e_p, e_l, cd_l) \\ modify(\lambda st.\ upSck(k_o, st)) \end{array}\right.$$

where $e' = ffold(init(ct), empty(address(e), sender(e), svalue(e)), dom(ct))$.

The main difference to internal method calls is that we do expect the method to not have any return expression but a $\bot$ instead. Consequently, we do not need to evaluate the return expression.

6.4 External Calls

External method invocations are again similar to external method calls:

$$stmt(External(ad, i, xe, val), e_p, e, cd) \overset{\text{def}}{=}$$

$$\textbf{do} \left[\begin{array}{l} gascheck(costs(External(ad, i, xe, val), e_p, e, cd)) \\ kad \leftarrow expr(ad, e_p, e, cd) \\ \llcorner\!-\!\lrcorner \begin{cases} \boxed{1} & \text{if } kad = (Simple(adv), Value(TAddr)) \wedge e_p(adv) = (ct, fb)_\bot \\ throw(Err) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1} = \textbf{do} \left[\begin{array}{l} kv \leftarrow expr(val, e_p, e, cd) \\ \llcorner\!-\!\lrcorner \begin{cases} \boxed{2} & \text{if } kv = (Simple(v), Value(t)) \wedge ct(i) = Method(fp, f, \bot)_\bot \\ \boxed{3} & \text{if } kv = (Simple(v), Value(t)) \wedge ct(i) = \bot \\ throw(Err) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{2} = \textbf{do} \left[\begin{array}{l} (e_l, cd_l, k_l, m_l) \leftarrow load(True, fp, xe, e_p, e', emptyStore, emptyStore, emptyStore, e, cd) \\ option\ Err \left[\begin{array}{l} \lambda st.\ transfer(address(e), adv, v, acc(st)) \\ \lambda acc.\ \textbf{do} \left[\begin{array}{l} (k_o, m_o) \leftarrow apply(\lambda st.\ (sck(st), mem(st))) \\ modify(\lambda st.\ upMem(m_l, upSck(k_l, upAcc(acc, st)))) \\ stmt(f, e_p, e_l, cd_l) \\ modify(\lambda st.\ upMem(m_o, upSck(k_o, st))) \end{array}\right. \end{array}\right. \end{array}\right.$$

$$\boxed{3} = option\ Err \left[\begin{array}{l} \lambda st.\ transfer(address(e), adv, v, acc(st)) \\ \lambda acc.\ \textbf{do} \left[\begin{array}{l} (k_o, m_o) \leftarrow apply(\lambda st.\ (sck(st), mem(st))) \\ modify(\lambda st.\ upMem(emptyStore, \\ \qquad\qquad upSck(emptyStore, upAcc(acc, st)))) \\ stmt(fb, e_p, empty(adv, address(e), v), cd) \\ modify(\lambda st.\ upMem(m_o, upSck(k_o, st))) \end{array}\right. \end{array}\right.$$

where $e' = ffold(init(ct), empty(adv, address(e), v), dom(ct))$.

There is, however, one notable difference to external method calls. For the case that we try to invoke a method which does not exist ($\boxed{3}$), we do indeed execute the contracts fallback function instead.

6.5 Transfer

The transfer statement is used to transfer funds from the current contract to another account. Its semantics is defined as follows:

$$stmt(Transfer(ad, ex), e_p, e, cd) \overset{\text{def}}{=}$$

$$\textbf{do} \left[\begin{array}{l} kv \leftarrow expr(ex, e_p, e, cd) \\ \llcorner\!\lrcorner \begin{cases} \boxed{1} & \text{if } kv = (Simple(v), Value(t)) \\ throw(Err) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1} = \textbf{do} \left[\begin{array}{l} kv' \leftarrow expr(ad, e_p, e, cd) \\ \llcorner\!\lrcorner \begin{cases} \boxed{1.1} & \text{if } kv' = (Simple(adv), Value(TAddr)) \\ throw(Err) & \text{otherwise} \end{cases} \end{array}\right.$$

$$\boxed{1.1} = option\ Err \left[\begin{array}{l} \lambda st.\ transfer(address(e), adv, v, acc(st)) \\ \lambda acc.\ \begin{cases} \boxed{1.1.1} & \text{if } e_p(adv) = (ct, f)_\perp \\ modify(\lambda st.\ upAcc(acc, st)) & \text{if } e_p(adv) = \perp \end{cases} \end{array}\right.$$

$$\boxed{1.1.1} = \textbf{do} \left[\begin{array}{l} (k_o, m_o) \leftarrow apply(\lambda st.\ (sck(st), mem(st))) \\ modify(\lambda st.\ upMem(emptyStore, upSck(emptyStore, upAcc(acc, st)))) \\ stmt(f, e_p, e', emptyStore) \\ modify(\lambda st.\ upMem(m_o, upSck(k_o, st))) \end{array}\right.$$

where $e' = ffold(init(ct), empty(adv, address(e), v), dom(ct))$.

First we compute the value *ex* to be transferred and the target address *ad*. Then, we transfer the desired funds from the account of the executing contract to the target account using function *transfer* discussed above. If the target address refers to a normal account we are done. However, if the target address refers to another contract then we need to invoke its fallback function.

Example 6.1 (Transfer). Let's assume we are given a contract `Receiver` containing a member variable `received` and fallback method which assigns value `True` to the `received` flag. In our model this would be represented by a procedure environment env_p with the following entry for the receiver contract:

$$env_p(\text{“}Receiver\text{”}) = [\text{“}received\text{”}, Var(STValue(TBool)), Id(\text{“}received\text{”}){=}True]$$

Moreover, let's assume that the received flag is currently set to `False` which is represented by a state *st* with the following entry for storage:

$$sto(st)(\text{“}Receiver\text{”})(\text{“}received\text{”}) = \text{“}False\text{”}$$

Finally, let's assume that we are given another contract `Sender` and both contracts, `Sender` and `Receiver` have a current balance of 100 wei[6]:

- $acc(st)(\text{“}Sender\text{”}) = \text{“}100\text{”}$
- $acc(st)(\text{“}Receiver\text{”}) = \text{“}100\text{”}$

Now, assume that the sender contract executes a *Transfer* statement to transfer 10 wei to the receiver contract. To this end we use an environment env_v with

$$address(env_v) = \text{“}Sender\text{”}$$

to indicate the currently executing contract and assume

$$stmt(Transfer(Address(\text{“}Receiver\text{”}), UInt(256, 10)), env_p, env_v, cd)(st) = Normal(st')$$

This indicates that the statement executed successfully and updated the state to st'. Inspecting the accounts of st' reveals that the execution updated the balances of the contracts accordingly:

- $acc(st')(\text{“}Sender\text{”})=\text{“}90\text{”}$
- $acc(st')(\text{“}Receiver\text{”})=\text{“}110\text{”}$

However, the transfer command also triggered the execution of the receiving contracts fallback function which changed its received flag to “*True*”:

$$sto(st')(\text{“}Receiver\text{”})(\text{“}received\text{”}) = \text{“}True\text{”}$$

7 PROVING TERMINATION

The formalization of our abstract gas model as well as the termination proof can be found in theory `Statements.thy`. As mentioned in Sect. 3.3.2 we use abstract cost functions (introduced in Eq. (`costs`$_e$) and Eq. (`costs`)), to represent the gas cost of a given expression/statement. The actual gas fees are computed on the level of the Ethereum byte code [55] and, moreover, are frequently updated. Thus, our Solidity formalization does not provide a built-in gas model trying to faithfully represent the actual gas model on the level of Ethereum bytecode. Rather, we allow for an external gas model to be added later on by providing actual definitions for these costs functions. However, to be able to ensure termination we do assume that the costs of certain expressions/statements is not zero (`statement_with_gas`):

$$\begin{aligned}
&0 < costs(Call(id, es), env_p, env, cd, st)\\
&0 < costs(ECall(ad, id, xe, val), env_p, env, cd, st)\\
&0 < costs(While(ex, stm), env_p, env, cd, st)\\
&0 < costs(Invoke(id, es), env_p, env, cd, st)\\
&0 < costs(External(ad, id, xe, val), env_p, env, cd, st)\\
&0 < costs(Transfer(ad, xe), env_p, env, cd, st)
\end{aligned} \tag{25}$$

To capture these requirements we use Isabelle's module concept, called locale [32]. Locales allow us to make use of these properties in proofs. These proofs are then valid for any concrete cost model (e.g., the one presented in Sect. 7.3) that satisfies these requirements. Technically, the user defining a new cost model needs to prove that their cost model is a so-called interpretation of our generic cost model. For this, the user has to prove only that their cost model satisfies these requirements.

Our generic cost model is not a limitation with respect to the actual cost model of Solidity: according to [55, Appendix G], the actual costs for any execution of these expressions/statements will be positive.

[6]Wei is the smallest denomination of ether—the cryptocurrency coin used on the Ethereum network. One ether = 10^{18} wei.

7.1 Measure function

To prove termination we need to provide an appropriate measure function which can be used to create a well-formed relation which decreases with every recursive call. To understand the construction of the measure function we need to understand a bit about how Isabelle represents mutual recursive functions. To eliminate the mutual dependencies, Isabelle internally creates a single function operating on the sum type of the different function specifications. Consequently, termination has to be proved simultaneously for both functions, by specifying a measure on the sum type.

The overall idea of the measure function is to use the lexicographic combination of gas costs and the number of data type constructors. To this end we first specify a function *mgas* over the sum type of our semantic functions:

$$mgas(x) \overset{\text{def}}{=} \begin{cases} gas(st) & \text{if } x = msel(mm, tp, loc, es, env_p, env, cd, st) \\ gas(st) & \text{if } x = ssel(tp, loc, es, env_p, env, cd, st) \\ gas(st) & \text{if } x = lexp(lv, env_p, env, cd, st) \\ gas(st) & \text{if } x = rexp(lv, env_p, env, cd, st) \\ gas(st) & \text{if } x = load(cp, fp, es, env_p, env, cd_o, k, m, e, c, st) \\ gas(st) & \text{if } x = expr(ex, env_p, env, cd, st) \\ gas(st) & \text{if } x = stmt(st, env_p, env, cd_o, k, m, e, c, st) \end{cases} \qquad (\texttt{mgas})$$

Note that this function simply extracts the gas costs of the parameter representing the state in the corresponding function.

The second function *msize* is defined as follows:

$$msize(x) \overset{\text{def}}{=} \begin{cases} size(es) & \text{if } x = msel(mm, tp, loc, es, env_p, env, cd, st) \\ size(es) & \text{if } x = ssel(tp, loc, es, env_p, env, cd, st) \\ size(lv) & \text{if } x = lexp(lv, env_p, env, cd, st) \\ size(lv) & \text{if } x = rexp(lv, env_p, env, cd, st) \\ size(es) & \text{if } x = load(cp, fp, es, env_p, env, cd_o, k, m, e, c, st) \\ size(ex) & \text{if } x = expr(ex, env_p, env, cd, st) \\ size(st) & \text{if } x = stmt(st, env_p, env, cd_o, k, m, e, c, st) \end{cases} \qquad (\texttt{msize})$$

This definition uses the function *size* which returns the size of the term using type specific definitions that are, for instance, provided by the function package and datatype package of Isabelle.

7.2 Termination

Now to verify termination of our semantic functions we need to verify two properties. First, we need to show that the lexicographic combination of our two measure functions is indeed well-founded:

Lemma 7.1 (Well founded). *The lexicographic combination of mgas and msize is well-founded.*

Next, we need to show that the measure function decreases with each recursive call. To this end we first prove the following lemma:

Lemma 7.2 (`msel_ssel_lexp_expr_load_rexp_stmt_dom_gas`).

$$msel(mm, tp, loc, es, env_p, env, cd)(st) = Normal(\_, st') \implies gas(st') \leq gas(st)$$
$$ssel(tp, loc, es, env_p, env, cd)(st) = Normal(\_, st') \implies gas(st') \leq gas(st)$$
$$lexp(lv, env_p, env, cd)(st) = Normal(\_, st') \implies gas(st') \leq gas(st)$$

$$\begin{aligned}
rexp(lv, env_p, env, cd)(st) = Normal(\_, st') &\implies gas(st') \leq gas(st)\\
load(cp, fp, es, env_p, env, cd_o, k, m, e, c)(st) = Normal(\_, st') &\implies gas(st') \leq gas(st)\\
expr(ex, env_p, env, cd)(st) = Normal(\_, st') &\implies gas(st') \leq gas(st)\\
stmt(st, env_p, env, cd_o, k, m, e, c)(st) = Normal(\_, st') &\implies gas(st') \leq gas(st)
\end{aligned}$$

Proof. The proof is by structural induction over Solidity statements/expressions. Some cases are trivial and do not need further discussion. For the remaining 28 cases the proofs proceed as follows: We construct a sequence of abstract states following the definition of the corresponding statement/expression. For each state we then show that it does not increase the amount of available gas. Thus, by transitivity of gas costs we can conclude that the overall statement does not increase the gas costs. □

We can then use this lemma and the assumptions about our cost functions (Equation 25) to prove the following lemma:

Lemma 7.3. *For each recursive call x' from a function x the following holds:*

- $mgas(x') < mgas(x)$
- $mgas(x') = mgas(x) \land msize(x')$

We now have everything to prove the following theorem:

Theorem 7.4 (`termination`). *The following functions are always defined:*

$msel(mm, tp, loc, es, env_p, env, cd)(st)$

$ssel(tp, loc, es, env_p, env, cd)(st)$

$lexp(lv, env_p, env, cd)(st)$

$rexp(lv, env_p, env, cd)(st)$

$load(cp, fp, es, env_p, env, cd_o, k, m, e, c)(st)$

$expr(ex, env_p, env, cd)(st)$

$stmt(st, env_p, env, cd_o, k, m, e, c)(st)$

Note that this result has several important practical consequences. First, it allows us to execute our functions and generate code. Second, it allows us to use induction to verify properties over each of the above functions, in particular *expr* and *stmt*.

7.3 A minimal cost model

To provide a concrete cost model for our semantics we just need to provide a definition for our cost functions which satisfy the conditions from Equation 25. Consider, for example the following definitions:

$$costs_{min}(stm, env_p, env, cd, st) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } stm = While \;\lor\; stm = Transfer \;\lor\; stm = Invoke \\ & \qquad\qquad \lor\; stm = External \\ 0 & \text{otherwise} \end{cases}$$

(`costs_min`)

$$costs_{min}(ex, env_p, env, cd, st) \overset{\text{def}}{=} \begin{cases} 1 & \text{if } stm = Call \;\lor\; stm = ECall \\ 0 & \text{otherwise} \end{cases}$$

(`costs_ex`)

The definitions obviously satisfy the conditions from Equation 25. Thus, this is a valid cost function to be used with our semantics.

8 CASE STUDY: VERIFIED BANKING

As shown in previous work [20], our semantics can be used to verify the correctness of tools which modify Solidity programs. However, it can also be used to verify invariants for individual contracts as demonstrated in the following case study.

To use our semantics to verify invariants for individual contracts we need to perform three steps:

- *Formalize the contract.* A contract is formalized as a finite mapping from an identifier to a corresponding member which can either be a storage variable or a method (`Member`). A method consists of a list of formal parameters, an optional return value, and a method body, which is formalized as a statement of our semantics.
- *Formalize the invariant.* The invariant is formalized as a HOL predicate over two parameters: an integer which represents the balance of the contract as well as a state which can be used to access the contract's private storage (`State`).
- *Verify the invariant.* To verify the invariant we need to prove one theorem for each method of our contract. The theorem assumes that the method is executed in a state which satisfies the invariant and shows that the invariant still holds in the state obtained after executing the method.

Note that the verification of methods which invoke methods from other contracts requires some considerations. Since we usually don't know the implementation of the method, we need to verify that the invariant is preserved for every possible implementation of the method. Thus, for such a case, we need to prove a corresponding lemma using structural induction over statements (lemma `msel_ssel_lexp_expr_load_rexp_stmt.induct`).

8.1 The problem

In the following we use the above methodology to verify a contract implementing a simple banking system (theory `Reentrancy.thy`).

The contract should allow users to deposit funds and withdraw them later on. A possible solution is given by the contract briefly discussed in the introduction and repeated in Listing 3. The contract

Listing 3: A simple banking contract.

```
contract Bank {
  mapping(address => uint256) balances;

  function deposit() public payable {
    balances[msg.sender] = balances[msg.sender] + msg.value;
  }

  function withdraw() public {
    msg.sender.transfer(balances[msg.sender]);
    balances[msg.sender] = 0;
  }
}
```

has one member variable `balances` to keep track of all the balances. Moreover, it provides two methods to deposit and withdraw funds. When a contract calls `deposit` with some funds, then the funds are transferred to the Bank contract and the amount is kept in `msg.value`. Thus, method `deposit` just adds the value to the balance of the calling contract to keep track of how much each contract contributed to the funds of the banking contract. A contract can call `withdraw` to get its funds back. To this end, the banking contract first returns the corresponding funds (Line 9) and then sets the callers internal balance to 0 (Line 10).

The contract in Listing 4 is a possible client contract for our bank contract, assuming our bank contract got address `0x438eacEBf3F2a1c3E8560277345E83ff228355bE` assigned during its deployment. Once deployed, we can send a transaction to invoke `deposit` method which will

Listing 4: A simple client contract.

```
contract Client {
  function deposit() public payable {
    0x438eacEBf3F2a1c3E8560277345E83ff228355bE.call.value(1 ether)(
      abi.encodeWithSignature("deposit()")
    );
  }

  function withdraw() public {
    0x438eacEBf3F2a1c3E8560277345E83ff228355bE.call(
      abi.encodeWithSignature("withdraw()")
    );
  }
}
```

then deposit 1 ether to the bank. To claim our money back we can send a transaction to invoke `withdraw` which just forwards the call to the bank to get its funds back.

In our example, contract Bank has a so-called *reentrancy* vulnerability. To this end, remember that transferring funds to a contract implicitly invokes the recipients fallback function. This behaviour can be exploited by an attacker as demonstrated by contract in Listing 5. Contract `Malicious` is similar to our client contract with one important difference: It uses its callback function to withdraw again as long as the bank has some funds left.

If we have such a contract deployed we can first use it to deposit 1 ether to the bank using `deposit`. This sets our internal balance in the bank contract to 1. Now we can invoke `withdraw` to claim our money back. However, Line 9 in the banking contract (recall Listing 1) not only transfers our money back but *afterwards* it also invokes our fallback function which calls `withdraw` again. This is done as long as the bank still owns more than 1 ether after which our fallback function terminates and the bank sets our balance to 0. However, in the meantime we were able to retrieve all the available funds from the bank.

8.2 Formalizing the contract

The problem with the banking contract (recall Listing 1) is that the money is being transferred *before* we adjust the callers balance. A simple solution to the problem is to just change the order of the two operations as done by the following contract shown in Listing 6. Here we first look up the current balance of the caller and remember it in a variable `bal`. Then, we adjust the caller's balance and only then we transfer the funds. If an attacker now tries to execute the same attack as before they would fail because at the time the fallback function is invoked their internal balance is already adjusted.

Listing 5: A malicious contract developed by an attacker.

```
contract Malicious {
  function deposit() public payable {
    0x438eacEBf3F2a1c3E8560277345E83ff228355bE.call.value(1 ether)(
      abi.encodeWithSignature("deposit()")
    );
  }

  function withdraw() public {
    0x438eacEBf3F2a1c3E8560277345E83ff228355bE.call(
      abi.encodeWithSignature("withdraw()")
    );
  }

  function () external payable {
    if (0x438eacEBf3F2a1c3E8560277345E83ff228355bE.balance > 1 ether) {
      0x438eacEBf3F2a1c3E8560277345E83ff228355bE.call(
        abi.encodeWithSignature("withdraw()")
      );
    }
  }
}
```

Listing 6: A modified banking contract.

```
contract MyToken {
  mapping(address => uint256) balances;

  function deposit() public payable {
    balances[msg.sender] = balances[msg.sender] + msg.value;
  }

  function withdraw() public {
    uint256 bal = balances[msg.sender];
    balances[msg.sender] = 0;
    msg.sender.transfer(bal);
  }
}
```

However, the question remains if the contract is indeed secure now or if it does expose other vulnerabilities. To answer this question, we first need to formalize the contract in our semantics.

Bank: **Identifier** ⇒ **Member** (bank)

$$\mathit{Bank} \overset{\text{def}}{=} \begin{cases} \text{“}\mathit{balances}\text{”} & \mapsto \quad \mathit{Var}(\mathit{STMap}(\mathit{TAddr}, \mathit{STValue}(\mathit{TUInt}(256)))) \\ \text{“}\mathit{deposit}\text{”} & \mapsto \quad \mathit{Method}([], \mathit{deposit}, \bot) \\ \text{“}\mathit{withdraw}\text{”} & \mapsto \quad \mathit{Method}([], \mathit{withdraw}, \bot) \end{cases}$$

The contract is formalized as a mapping from identifiers to corresponding members. While “*balances*” refers to a variable, “*deposit*” and “*withdraw*” refer to methods with body “deposit” and “withdraw”, respectively.

Deposit is defined as a simple assignment to the member “*balances*”:

$$\mathit{deposit} \overset{\text{def}}{=} \mathit{Ref}(\text{“}\mathit{balances}\text{”}, [\mathit{Sender}]) = L(\mathit{Ref}(\text{“}\mathit{balances}\text{”}, [\mathit{Sender}])) + \mathit{Value} \qquad (\texttt{deposit})$$

Withdraw, on the other hand, is defined as a block statement.

$$\mathit{withdraw} \overset{\text{def}}{=} \mathit{Block}((\text{“}\mathit{bal}\text{”}, \mathit{Value}(\mathit{TUInt}(256)), L(\mathit{Ref}(\text{“}\mathit{balances}\text{”}, [\mathit{Sender}]))), \mathit{comp}) \qquad (\texttt{keep})$$

$$\mathit{comp} \overset{\text{def}}{=} \mathit{Ref}(\text{“}\mathit{balances}\text{”}, [\mathit{Sender}]) = \mathit{UInt}(256, 0); \mathit{transfer} \qquad (\texttt{comp})$$

$$\mathit{transfer} \overset{\text{def}}{=} \mathit{Transfer}(\mathit{Sender}, L(\mathit{Id}(\text{“}\mathit{bal}\text{”}))) \qquad (\texttt{transfer})$$

8.3 Formalizing the Invariant

To verify our contract we verify that the following invariant is preserved by both methods of the banking contract.

$$\mathit{inv}(\mathit{bal}, \mathit{ac}, s) \overset{\text{def}}{=} \mathit{ac} - \mathit{sum}(s) \geq \mathit{bal} \wedge \mathit{bal} \geq 0 \wedge \mathit{pos}(s) \qquad (\texttt{frame\_def})$$

$$\mathit{sum}(s) \overset{\text{def}}{=} \sum_{\{(ad,x) \ \mid \ s(ad+\text{“.”}+\text{“}balances\text{”})=x_\bot\}} \lceil x \rceil \qquad (\texttt{SUMM})$$

$$\mathit{pos}(s) \overset{\text{def}}{=} \forall ad, x.\ s(ad + \text{“.”} + \text{“}\mathit{balances}\text{”}) = x_\bot \implies \lceil x \rceil \geq 0 \qquad (\texttt{POS})$$

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

8.4 Verifying the Invariant

As discussed above, Solidity implicitly triggers the call of a so-called fallback method whenever we transfer money to a contract. In particular if another contract calls `withdraw`, this triggers an implicit call to the callee's fallback method. Since we do not know all potential contracts which call `withdraw`, we need to verify our invariant for all possible Solidity programs.

Thus, we first prove that the invariant is preserved by every Solidity program which is not executed in the context of our own contract. To this end we verified corresponding lemmata for all our semantic functions. The one for *expr* is as follows.

Lemma 8.1 (`secure-expr`). *Let env_p be a contract environment which assigns contract Bank to address “Bank”:*

$$env_p(\text{“}Bank\text{”}) = (Bank, Skip)_\bot$$

Moreover, let env be a variable environment, such that the address of the executing contract is different from “Bank”:

$$address(env) \neq \text{“}Bank\text{”}$$

Finally, let st be a state which satisfies the invariant for a certain balance bal:

$$\exists s.\ sto(st)(\text{“Bank”}) = s_\bot \ \wedge\ inv(bal, \lceil acc(st)(\text{“Bank”})\rceil, s)$$

Then, the invariant is preserved by the semantics of an expression ex evaluated with an arbitrary calldata cd:

$$\forall st'.\ expr(ex, env_p, env, cd)(st) = Normal(\_, st') \\ \implies \exists s.\ sto(st')(\text{“Bank”}) = s_\bot \ \wedge\ inv(bal, \lceil acc(st')(\text{“Bank”})\rceil, s)$$

Proof. The proof is by structural induction over Solidity expressions. For the non-trivial cases we expand the definition of the expression and construct a sequence of abstract states following the definition of the expression. We then show that the invariant is preserved for each of the intermediate states. □

We verified similar lemmata for the remaining semantic functions: *msel*, *ssel*, *lexp*, *rexp*, and *load*. In particular, we have a similar result for *stmt*. However, for statements we do also need to consider the case in which the currently executing contract is the banking contract. Indeed, the invariant can be violated while executing the `withdraw` method as long as it is established again after execution. To this end, we verified different lemmata about pre- and corresponding postconditions for the different statements of the method.

For the actual transfer we verified the following lemma:

Lemma 8.2 (`secure-transfer`). *Let env_p be a contract environment as defined in Theorem 8.1. Moreover, let env be a variable environment, such that the currently executing address is the bank, the sender is not the bank, and variable “bal” refers to location x on the stack:*

$$\begin{aligned} address(env) &= \text{“Bank”} \\ sender(env) &\neq address(env) \\ denvalue(env)(\text{“bal”}) &= (Value(TUInt(256)), Stackloc(x))_\bot \end{aligned}$$

Finally, let st be a state in which the value val, stored in stack at location x, deduced from the balance of the bank satisfies the invariant:

$$\begin{aligned} & accessStore(x, sck(st)) = Simple(val)_\bot \\ & sto(st)(\text{“Bank”}) = s_\bot \\ & inv(bal, \lceil acc(st)(\text{“Bank”})\rceil - \lceil val\rceil, s) \end{aligned} \tag{26}$$

Then, executing transfers leads to a state which satisfies again the invariant:

$$\forall st'.\ stmt(transfer, env_p, env, cd)(st) = Normal((), st') \\ \implies \exists s.\ sto(st')(\text{“Bank”}) = s_\bot \ \wedge\ inv(bal, \lceil acc(st')(\text{“Bank”})\rceil, s)$$

Proof. First, we need to show that transferring of the funds from a state which satisfies the preconditions indeed establishes the invariant. However, since the precondition requires that we are in a state in which the invariant holds for a deduced banking balance and transfer actually reduces the banks balance by the very same amount, the invariant must hold afterwards.

In addition, the transfer statement invokes the execution of the fallback method of the sender, and we need to show that for all possible implementations of this fallback method, the invariant will not be violated. However, this follows from the fact that the external contract does not have direct access to the bank's private store and balance, and can only modify it indirectly by calling the methods provided by the bank. However, these are already shown to preserve the invariant. □

This lemma provides us with a set of preconditions that need to hold before transferring the funds to ensure that the invariant holds afterwards. Since the transfer command reduces the funds available to our banking contract, the precondition requires that we have a surplus corresponding to the value stored in variable *bal* on the stack (Equation 26).

The lemma for the composition uses the above lemma to establish preconditions for the compound statement:

LEMMA 8.3 (`secure-composition`). *Let env_p be a contract environment as defined in Theorem 8.1 and Theorem 8.2. Moreover, let env be a variable environment as defined in Theorem 8.2 but with the additional requirement that variable "balance" refers to the storage location "balance" which contains the contracts internal client balances:*

$$denvalue(env)(\text{“balance”}) = Storage(STMap(TAddr, STValue(TUInt(256))), Storeloc(\text{“balance”}))_\perp$$

Finally, let st be a state in which the stack location x contains the internal balance of the message sender:

$$accessStore(x, sck(st)) = Simple(accessStorage(TUInt(256), sender(env) + \text{“.”} + \text{“balance”}, s))_\perp \tag{27}$$

and which satisfies the invariant:

$$sto(st)(\text{“Bank”}) = s_\perp \ \wedge \ inv(bal, \lceil acc(st)(\text{“Bank”})\rceil, s)$$

Then, executing the composed statement leads to a state which satisfies again the invariant:

$$\forall st'.\ stmt(comp, env_p, env, cd)(st) = Normal((), st') \implies \exists s.\ sto(st')(\text{“Bank”}) = s_\perp \ \wedge \ inv(bal, \lceil acc(st')(\text{“Bank”})\rceil, s)$$

To be able to establish the invariant after executing the composed statement, we have to ensure that the invariant already holds before. Moreover, we need to ensure that variable *bal* on the stack indeed contains the value of the balance of the caller from the contracts internal balances (Equation 27).

We can now use the result for the composition to establish the preconditions for the withdraw method as a whole:

LEMMA 8.4 (`secure-withdraw`). *Let env_p be a contract environment as defined in Theorem 8.1 and Theorem 8.2. Moreover, let env be a variable environment, such that the currently executing address is the bank, the sender is not the bank, and variable "balance" refers to the storage location "balance" which contains the contracts internal client balances:*

$$\begin{aligned} address(env) &= \text{“Bank”} \\ sender(env) &\neq address(env) \\ denvalue(env)(\text{“balance”}) &= Storage(STMap(TAddr, STValue(TUInt(256))), \\ &\qquad Storeloc(\text{“balance”}))_\perp \end{aligned}$$

Finally, let st be a state which satisfies the invariant for a certain balance bal:

$$\exists s.\ sto(st)(\text{“Bank”}) = s_\perp \ \wedge \ inv(bal, \lceil acc(st)(\text{“Bank”})\rceil, s)$$

Then, executing withdraw leads to a state which satisfies again the invariant:

$$\forall st'.\ stmt(withdraw, env_p, env, cd)(st) = Normal((), st')$$

$$\Longrightarrow\ \exists s.\ sto(st')(\text{“Bank”}) = s_{\perp}\ \wedge\ inv(bal, \lceil acc(st')(\text{“Bank”})\rceil, s)$$

In particular, the lemma requires that the invariant holds before the execution. In addition, it requires that the variable environment contains the correct reference to the contracts balances. This second condition is established automatically when calling the withdraw method.

Thus, we are now able to prove the following theorem about our banking contract:

Theorem 8.5 (`final1`). *Let env_p be a contract environment which assigns contract Bank to address “Bank”:*

$$env_p(\text{“Bank”}) = (Bank, Skip)_{\perp}$$

Moreover, let env be a variable environment, such that the address of the executing contract is different from “Bank”:

$$address(env) \neq \text{“Bank”}$$

Finally, let st be a state which satisfies the invariant for a certain balance bal:

$$\exists s.\ sto(st)(\text{“Bank”}) = s_{\perp}\ \wedge\ inv(bal, \lceil acc(st)(\text{“Bank”})\rceil, s)$$

Then, the invariant is preserved by executing method “withdraw” of contract “Bank”:

$$\forall st'.\ stmt(External(Address(\text{“Bank”}), \text{“withdraw”}, [], val), env_p, env, cd, st) = Normal((), st')$$
$$\Longrightarrow\ \exists s.\ sto(st')(\text{“Bank”}) = s_{\perp}\ \wedge\ inv(bal, \lceil acc(st')(\text{“Bank”})\rceil, s)$$

The theorem states that method `withdraw` indeed preserves our invariant. We verified a similar result for the `deposit` method as well (`final2`).

Note that since the contract itself as well as unknown contracts may depend on each other, all the lemmata are encoded in a single lemma which is then proved by mutual induction. The final result is then given in terms of two corollaries for the corresponding methods of our contract.

Note also that in an earlier version of the proof we tried to verify a stronger invariant given by

$$\exists s.\ sto(st)(\text{“Bank”}) = s_{\perp} \wedge \lceil acc(st)(\text{“Bank”})\rceil = sum(s) \geq bal$$

In particular we thought that the sum of internal balances will be equal to the funds of the contract. However, during verification we found that this property does not necessarily hold. In particular another contract may just “gift” some money to the banking contract without going through the `deposit` method. In such a case, the gift will not be recognized in the contracts internal balances, and we needed to modify the invariant. While the current version of the invariant does not guarantee that no money is gifted to the contract it does, however, guarantee that no money can be stolen which is indeed the desired property.

The second thing to note is that we were not able to verify that the difference is indeed constant. During verification, it turned out that this is not the case since in the fallback method a contract could simply send us additional money without calling “deposit”. In such a case the difference would change. In particular, it would grow. However, we were able to verify that the difference does never shrink which is what we actually want to ensure.

9 DISCUSSION

In the following, we discuss briefly various aspects of our semantics, respectively, of design decisions that we took when formalising it in Isabelle/HOL.

Supported Language Features. In Appendix A we provide a detailed overview of all supported and unsupported Solidity language features. In summary, our formalization supports the following key features of the language:

- *Fixed-size integer types* of various lengths and arithmetic with support for overflows.
- *Domain-specific primitives*, such as money transfer or balance queries.
- *Different types of stores*, such as storage, memory, calldata, and stack.
- *Complex data types*, such as hash-maps and arrays.
- *Assignments with different semantics*, depending on the location of the involved data types.
- An extendable *gas model* which ensures termination.
- Internal and external *method calls* with the ability to send funds with external calls.

However, there are also some important features which are not supported yet. Most importantly, our formalization does not support the notion of checked arithmetic. This feature is now the default in newer versions of Solidity and thus it is becoming increasingly important and should be added in future work. In addition, our formalization does not support user defined types in the form of structs. Although this can be encoded using our notion of array (each field becomes an entry in the array), it would be nice to have a more explicit notion of structs. Moreover, we do not support custom exception handling mechanisms. However, our formalization is based on an exception monad and it should not be too difficult to add additional types of exceptions and handlers. Finally, we do not support the creation of new contracts at runtime as well as inheritance. These features are not so common in practice yet and it remains to see if they are required.

Cost Model. As many other languages for writing contracts on blockchains, Solidity is equipped with a cost model [55, Appendix G]. This cost model is defined in terms of operations of the Solidity bytecode. Thus, we cannot provide a precise formalisation of the actual cost model, as the actual costs do depend on how a compiler translates (and optimises) Solidity into bytecode. Thus, we opted for a pluggable cost model on the level of the Solidity source language: we only require that each command does cost a non-negative amount of Ether (this is justified by the actual costs specified in [55, Appendix G]). A user of our framework is able to plug in a cost model that suits their needs, as long as it satisfies this minimal requirement.

Monadic Representation. Compared to earlier versions of our work [37], we use a monadic representation in the version described in this paper. In our experience, this significantly improved the readability of the formal semantics. A drawback is that Isabelle is no longer able to prove termination of the semantic function automatically. Hence, we needed to provide a manual proof for this (see Sect. 7).

Executability of our Semantics. When designing our semantics for Solidity, we took care that the semantics is executable. This gives us several advantages: first, properties can be proven by the simplifier, essentially using symbolic execution. For more complex properties, we can make use of a technique called normalisation by evaluation that, internally, relies on being able to generate executable code [27]. Second, we can explore our semantics by evaluating it for ground terms within Isabelle, using Isabelle' value-statement. Third, we can generate a trustworthy interpreter for our formal semantics using Isabelle's code generator [27]. The latter can be used as test oracle for compliance testing (see discussion below).

Updates to the Solidity Language. The Solidity language is under frequent development and, hence, new versions of the language specification are published frequently. This is not only a challenge for formalisation efforts, this is also a general challenge for developers of tools for Solidity and for Solidity developers themselves. Still, this raises questions how we can update our semantics and how we can ensure that our semantics complies to a certain version of the Solidity language. While we do not have a silver bullet in terms of updating the formal semantics, our conservative approach guarantees that none of the changes we make during an update does endanger the consistency of our semantics. This is a big advantage over approaches that define the semantics axiomatically. For the latter, see the next paragraph.

Compliance of the Formal Semantics. To ensure the compliance of our semantics to the actual implementation of Solidity, we developed a grammar-based fuzzing framework [38]. For this, we leverage that our semantics is executable: in more detail, we generate a trustworthy interpreter of Solidity using Isabelle's code generator for Haskell. This stand-alone interpreter is used as test-oracle, i.e., we execute a test-case—a Solidity program—on both the Ethereum blockchain and our interpreter and compare the results. If both yield the same result state (starting from the same initial state), our semantics complies to the implementation. If they differ, we found a divergence, and we need to update our semantics. Again, given our conservative approach of defining the Solidity semantics, such updates do not endanger the consistency of the semantics. Finally, for generating test cases, we use a grammar-based fuzzing approach based on a type-enriched grammar of Solidity. For more details, we refer the reader elsewhere [38].

Secure Transfer. Newer versions of Solidity impose restrictions on the amount of Gas that can be used in a fallback function. Typically, the provided Gas is insufficient to call back into the original contract, thus preventing reentrancy attacks. Although this could be verified from our semantics by providing an appropriate Gas model, Gas limits are subject to change, and there's no assurance that the Gas available for fallback functions won't be adjusted in the future. Consequently, contracts verified without relying on a specific Gas model are more future-proof.

10 RELATED WORK

The work presented in this paper is about the formalization of Solidity in Isabelle. Thus, related work can be found in two different areas: First, work related to the formalization of Solidity and work related to the formalization of programming languages using Isabelle. We discuss other works formalizing Solidity in Isabelle at the end of this section.

10.1 Formalizations of Solidity

As outlined by Almakhour et al. [3] and Tolmach et al. [52], there is a growing amount of research investigating the formalization of smart contracts. While most of this work has focussed on the formalization of low-level bytecode, there exists also work which focusses on the formalization of Solidity. Early work in this area was done by Bhargavan et al. [11] which describe an approach to map a Solidity contract to F* where it can then be verified. Mavridou et al. [41], provide an approach based on FSolidM [40], in which a Solidity smart contract is modelled as a state machine to support model checking of common security properties. TinySol [8] and Featherweight Solidity[18], on the other hand, are two calculi formalizing some core features of Solidity. Crosara et al. [19] 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, Zakrzewski [58] describes a big-step semantics of a small subset of Solidity and Yang and Moreover, Hajdu and Jovanovic [28, 29], provide a formalization of Solidity in terms of a simple SMT-based intermediate language which they evaluate on a set of manually developed tests. In addition, Jiao et al. [30, 31], provide a formalization of Solidity in $\mathbb{K}$ with a rigorous evaluation using the Solidity compiler test set. Finally, Singh et al. [50] describe a formalization of Solidity in Event-B. All of these works use an axiomatic approach for defining the formal semantics of Solidiy.

In contrast, we use a conservative embedding into Isabelle/HOL, which ensures the consistency of our semantics "by construction".

10.2 Formalization the Semantics of Programming Languages in Isabelle

Formalizing programming languages or specification languages in Isabelle is by no means a new technique. Over the substantial body of languages and tools have been developed along this line, which have seen substantial applications—we cite only the current flagships of this development Isabelle/SIMPL [49] and the seL4 verification project [34]. As our work, most embeddings of programming languages in Isabelle/HOL, e.g., [35, 44, 49], are deep embeddings (for a discussion of deep and shallow embeddings, we refer the reader to [4, 12]). Also, using a monadic representation of stateful computations is common. For example, the AutoCorres [13, 26] (used in the seL4 verification project [34]) provides an abstraction of C code using monads, and also Clean [53], as an example of a shallow embedding, does use monads for modelling stateful computations.

10.3 Formalization of Solidity in Isabelle

To the best of our knowledge there is only one other formalization of Solidity in Isabelle by Ribeiro et al. [47]. The authors adapt the Simpl language [48] to formalize a subset of Solidity in Isabelle/HOL. SOLI supports the following language features:

- `Skip`: the empty program
- `Upd`: execute a state-update function
- `Seq`: sequential composition
- `If`, `While`: conditionals and loops
- `Dyncom`: receives a state and allows to write statements which are state dependent.
- `Call`: calling a procedure
- `Revert`, `Handle`: Throw and handle exceptions
- `Require`: Solidity exceptions
- `Init`: State reversion

Compared to our work, SOLI is quite low-level and rather an intermediate language than a direct formalization of Solidity. Indeed, while it supports several features which are not provided by Solidity (Upd, Dyncom), it does not provide explicit support for Solidity-specific language features, such as different types of stores, a notion of Gas, fallback methods, external vs. internal functions, etc. Another difference to our work is that their semantics seems to be not executable and therefore difficult to evaluate. On the other hand, we considered it important to have an executable semantics that can be evaluated against the reference implementation.

11 CONCLUSION

In this paper we presented a formal semantics of a subset of Solidity with support for the following language features:

- *Fixed-size integer types* of various lengths and corresponding arithmetic with support for overflows.
- *Domain-specific primitives*, such as money transfer or balance queries.
- *Different types of stores*, such as storage, memory, calldata, and stack.
- *Complex data types*, such as hash-maps and arrays.
- *Assignments with different semantics*, depending on the location of the involved data types.
- Internal and external *method calls* with the ability to send funds with external calls.

Our semantics provides also an abstract *gas model* which ensures termination. This abstract model can be instantiated by providing concrete cost functions to support verification of gas-related aspects.

Our semantics is implemented in Isabelle as a deep embedding in Isabelle/HOL and can be symbolically executed. Since symbolic execution can sometimes be slow, we also provide code generation to execute the semantics in Haskell. To ensure that our semantics describes the behaviour of the actual Solidity implementation faithfully we developed a test framework to automatically test our semantics against a reference implementation of Solidity. The framework uses grammar-based fuzzing to generate random Solidity programs which are then executed with our semantics and on the actual blockchain to compare the outcome and detect potential deviations.

As demonstrated by our case study, our semantics can indeed be used to verify the correctness of smart contracts. However, the case studies also revealed some limitations of our semantics. In particular, reasoning from basic definitions can be tedious and verification requires quite some manual effort. Thus, future work should focus on improving the support for verification by developing calculi and verification condition generators for our semantics.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their careful reading and their comments and suggestions to improve earlier versions of this paper. Moreover, we thank Wolfgang Ahrendt and Richard Bubel for the useful discussions about the intricacies of the semantics of Solidity. This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/X027619/1].

REFERENCES

[1] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H Schmitt, and Mattias Ulbrich. 2016. *Deductive software verification–the KeY book*. Vol. 10001. Springer.

[2] Wolfgang Ahrendt and Richard Bubel. 2020. Functional Verification of Smart Contracts via Strong Data Integrity. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, Tiziana Margaria and Bernhard Steffen (Eds.). Springer, 9–24.

[3] Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. 2020. Verification of smart contracts: A survey. *Pervasive and Mobile Computing* 67 (2020), 101227. https://doi.org/10.1016/j.pmcj.2020.101227

[4] Catia M. Angelo, Luc J. M. Claesen, and Hugo De Man. 1994. Degrees of Formality in Shallow Embedding Hardware Description Languages in HOL. In *Higher Order Logic Theorem Proving and Its Applications (HUG)*, Jeffrey J. Joyce and Carl-Johan H. Seger (Eds.), Vol. 780. 89–100. https://doi.org/10.1007/3-540-57826-9_127

[5] Solidity Authors. 2021. Solidity documentation. https://docs.soliditylang.org/en/v0.5.16/. Accessed: 2021-05-01.

[6] Solidity Authors. 2024. Solidity Developer Survey 2023 Results. https://soliditylang.org/blog/2024/04/03/solidity-developer-survey-2023-results/

[7] T. Bahrynovska. 2017. History of Ethereum Security Vulnerabilities, Hacks and Their Fixes. https://web.archive.org/web/20190628084427/https://applicature.com/blog/blockchain-technology/history-of-ethereum-security-vulnerabilities-hacks-and-their-fixes

[8] Massimo Bartoletti, Letterio Galletta, and Maurizio Murgia. 2019. A Minimal Core Calculus for Solidity Contracts. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, Cristina Pérez-Solà, Guillermo Navarro-Arribas, Alex Biryukov, and Joaquin Garcia-Alfaro (Eds.). Springer, 233–243.

[9] Stefan Berghofer and Markus Wenzel. 1999. Inductive Datatypes in HOL — Lessons Learned in Formal-Logic Engineering. In *TPHOLs*, Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin (Eds.). Springer, 19–36.

[10] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. 2013. Keccak. In *EUROCRYPT*, Thomas Johansson and Phong Q. Nguyen (Eds.). Springer, 313–314.

[11] Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. 2016. Formal Verification of Smart Contracts: Short Paper. In *Programming Languages and Analysis for Security* (Vienna, Austria) *(PLAS)*. ACM, 91–96. https://doi.org/10.1145/2993600.2993611

[12] Richard Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. 1993. Experience with embedding hardware description languages in HOL. In *the International Conference on Theorem Provers*

in Circuit Design: Theory, Practice and Experience (22–24 June 1993) *(IFIP Transactions, Vol. A-10)*, Victoria Stavridou, Thomas F. Melham, and Raymond T. Boute (Eds.). 129–156.

[13] Matthew Brecknell, David Greenaway, Johannes Hölzl, Fabian Immler, Gerwin Klein, Rafal Kolanski, Japheth Lim, Michael Norrish, Norbert Schirmer, Salomon Sickert, Thomas Sewell, Harvey Tuch, and Simon Wimmer. 2024. AutoCorres2. *Archive of Formal Proofs* (April 2024). https://isa-afp.org/entries/AutoCorres2.html, Formal proof development.

[14] Lukas Bulwahn, Alexander Krauss, Florian Haftmann, Levent Erkök, and John Matthews. 2008. Imperative Functional Programming with Isabelle/HOL. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 134–149.

[15] Gertrude Chavez-Dreyfuss. 2016. Sweden tests blockchain technology for land registry.

[16] CipherTrace. 2021. *Cryptocurrency Crime and Anti-Money Laundering Report.* Technical Report. https://ciphertrace.com/cryptocurrency-crime-and-anti-money-laundering-report-august-2021/

[17] David Cock, Gerwin Klein, and Thomas Sewell. 2008. Secure Microkernels, State Monads and Scalable Refinement. In *Theorem Proving in Higher Order Logics*, Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 167–182.

[18] Silvia Crafa, Matteo Di Pirro, and Elena Zucca. 2020. Is Solidity Solid Enough?. In *Financial Cryptography and Data Security*, Andrea Bracciali, Jeremy Clark, Federico Pintore, Peter B. Rønne, and Massimiliano Sala (Eds.). Springer, 138–153.

[19] Marco Crosara, Gabriele Centurino, and Vincenzo Arceri. 2019. Towards an Operational Semantics for Solidity. In *VALID*, Jos van Rooyen, Samuele Buro, Marco Campion, and Michele Pasqua (Eds.). IARIA, 1–6.

[20] DM and A.D. Brucker. 2021. A Denotational Semantics of Solidity in Isabelle/HOL. In *SEFM*.

[21] A. Azaria et al. 2016. Medrec: Using Blockchain for Medical Data Access and Permission Management. In *OBD*.

[22] G. Batra et al. 2019. Blockchain 2.0: What's in store for the two ends?

[23] J. Mendling et al. 2018. Blockchains for Business Process Management – Challenges and Opportunities. (2018).

[24] Gartner. 2019. Forecast: Blockchain Business Value, Worldwide, 2017-2030.

[25] D. Goodin. 2021. Really stupid "smart contract" bug let hackers steal $31 million in digital coin. https://arstechnica.com/information-technology/2021/12/hackers-drain-31-million-from-cryptocurrency-service-monox-finance/

[26] David Greenaway, June Andronick, and Gerwin Klein. 2012. Bridging the Gap: Automatic Verified Abstraction of C. In *Interactive Theorem Proving*, Lennart Beringer and Amy Felty (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 99–115.

[27] Florian Haftmann and Lukas Bulwahn. 2023. Code generation from Isabelle/HOL theories. http://isabelle.in.tum.de/doc/codegen.pdf

[28] Ákos Hajdu and Dejan Jovanovic. 2019. solc-verify: A Modular Verifier for Solidity Smart Contracts. In *VSTTE (LNCS, Vol. 12031)*, Supratik Chakraborty and Jorge A. Navas (Eds.). Springer, 161–179. https://doi.org/10.1007/978-3-030-41600-3_11

[29] Ákos Hajdu and Dejan Jovanovic. 2020. SMT-Friendly Formalization of the Solidity Memory Model. In *ESOP (LNCS, Vol. 12075)*, Peter Müller (Ed.). Springer, 224–250. https://doi.org/10.1007/978-3-030-44914-8_9

[30] Jiao Jiao, Shuanglong Kan, Shang-Wei Lin, David Sanan, Yang Liu, and Jun Sun. 2020. Semantic understanding of smart contracts: executable operational semantics of Solidity. In *SP*. IEEE, 1695–1712.

[31] Jiao Jiao, Shang-Wei Lin, and Jun Sun. 2020. A Generalized Formal Semantic Framework for Smart Contracts. In *FASE*, Heike Wehrheim and Jordi Cabot (Eds.). Springer, 75–96.

[32] Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. 1999. Locales A Sectioning Concept for Isabelle. In *Theorem Proving in Higher Order Logics*, Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 149–165.

[33] J. Kelly. 2016. Banks adopting blockchain 'dramatically faster' than expected: IBM.

[34] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. seL4: formal verification of an operating-system kernel. *Commun. ACM* 53, 6 (2010), 107–115. https://doi.org/10.1145/1743546.1743574

[35] Peter Lammich and Simon Wimmer. 2019. IMP2 – Simple Program Verification in Isabelle/HOL. *Archive of Formal Proofs* (January 2019). https://isa-afp.org/entries/IMP2.html, Formal proof development.

[36] Defi Llama. 2024. TVL breakdown by Smart Contract Language. https://defillama.com/languages

[37] Diego Marmsoler and Achim D. Brucker. 2021. A Denotational Semantics of Solidity in Isabelle/HOL. In *Software Engineering and Formal Methods (SEFM)*, Radu Calinescu and Corina Pasareanu (Eds.). Springer-Verlag, Heidelberg. https://www.brucker.ch/bibliography/abstract/marmsoler.ea-solidity-semantics-2021

[38] Diego Marmsoler and Achim D. Brucker. 2022. Conformance Testing of Formal Semantics using Grammar-based Fuzzing. In *TAP 2022: Tests And Proofs*, Laura Kovacs and Karl Meinke (Eds.). Springer-Verlag, Heidelberg. https://www.brucker.ch/bibliography/abstract/marmsoler.ea-conformance-2022

[39] Diego Marmsoler and Achim D. Brucker. 2022. 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.

[40] Anastasia Mavridou and Aron Laszka. 2018. Tool Demonstration: FSolidM for Designing Secure Ethereum Smart Contracts. In *Principles of Security and Trust*, Lujo Bauer and Ralf Küsters (Eds.). Springer, 270–277.

[41] Anastasia Mavridou, Aron Laszka, Emmanouela Stachtiari, and Abhishek Dubey. 2019. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security*, Ian Goldberg and Tyler Moore (Eds.). Springer, 446–465.

[42] S. Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. (2008).

[43] BBC News. 2021. Hackers steal $600m in major cryptocurrency heist. https://www.cnbc.com/2021/08/23/poly-network-hacker-returns-remaining-cryptocurrency.html

[44] Tobias Nipkow. 1998. Winskel is (almost) Right: Towards a Mechanized Semantics Textbook. 10, 2 (1998), 171–186. https://doi.org/10.1007/s001650050009

[45] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. 2002. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS, Vol. 2283. Springer.

[46] Daniel Perez and Ben Livshits. 2021. Smart Contract Vulnerabilities: Vulnerable Does Not Imply Exploited. In *USENIX Security*. USENIX Association.

[47] Maria Ribeiro, Pedro Adão, and Paulo Mateus. 2020. *Formal Verification of Ethereum Smart Contracts Using Isabelle/HOL*. Springer International Publishing, Cham, 71–97. https://doi.org/10.1007/978-3-030-62077-6_7

[48] Norbert Schirmer. 2006. *Verification of Sequential Imperative Programs in Isabelle/HOL*. Ph. D. Dissertation. Technische Universität München.

[49] Norbert Schirmer. 2008. A Sequential Imperative Programming Language Syntax, Semantics, Hoare Logics and Verification Environment. *Archive of Formal Proofs* (Feb. 2008). http://afp.sf.net/entries/Simpl.shtml, Formal proof development.

[50] Neeraj Kumar Singh, Akshay M. Fajge, Raju Halder, and Md. Imran Alam. 2023. Chapter 8 - Formal verification and code generation for solidity smart contracts. In *Distributed Computing to Blockchain*, Rajiv Pandey, Sam Goundar, and Shahnaz Fatima (Eds.). Academic Press, 125–144. https://doi.org/10.1016/B978-0-323-96146-2.00028-0

[51] Christian Sternagel and René Thiemann. 2014. Certification Monads. *Archive of Formal Proofs* (Oct. 2014). https://www.isa-afp.org/entries/Certification_Monads.shtml. Formal proof development.

[52] Palina Tolmach, Yi Li, Shang-Wei Lin, Yang Liu, and Zengxiang Li. 2021. A Survey of Smart Contract Formal Specification and Verification. *ACM Comput. Surv.* 54, 7, Article 148 (jul 2021), 38 pages. https://doi.org/10.1145/3464421

[53] Fréderic Tuong and Burkhart Wolff. 2019. Clean – An Abstract Imperative Programming Language and its Theory. *Archive of Formal Proofs* (Oct. 2019). https://isa-afp.org/entries/Clean.html, Formal proof development.

[54] Philip Wadler. 1993. Monads for functional programming. In *Program Design Calculi*, Manfred Broy (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–264.

[55] Gavin Wood. 2021. *Ethereum: A Secure Decentralised Generalised Transation Ledger (Version 2021-04-21).* Technical Report. Ethereum.

[56] YCharts.com. 2022. Ethereum Transactions Per Day.

[57] B. Yurcan. 2016. How Blockchain Fits into the Future of Digital Identity.

[58] Jakub Zakrzewski. 2018. Towards Verification of Ethereum Smart Contracts. In *VSTTE (LNCS, Vol. 11294)*, Ruzica Piskac and Philipp Rümmer (Eds.). Springer, 229–247. https://doi.org/10.1007/978-3-030-03592-1_13

A SOLIDITY LANGUAGE FEATURES

In the following, we provide an overview of the language features of Solidity and their formalization. The list of features follows the official language specification[7]. A feature which is supported is marked by a ☑symbol. If it is not currently supported it is marked by a ☒symbol. If it is partially supported it is marked by a □ symbol.

- Structure of a Contract
  ☑ State Variables
  ☑ Functions
  ☒ Function Modifiers
  ☒ Events
  ☒ Errors

[7]https://docs.soliditylang.org/en/v0.8.26/

  - ☒ Struct Types
  - ☒ Enum Types
- Types
  - ☐ Value Types
    - ☑ Signed and unsigned integers from 8-256 bit
    - ☑ Address type
    - ☒ Contract Types
    - ☐ Others: some basic types are missing
  - ☐ Reference Types
    - ☐ Memory: arrays but not structs
    - ☐ Storage: arrays but not structs
    - ☐ Calldata: arrays but not structs
  - ☑ Mapping types
  - ☐ Operators: some operators are missing
  - ☐ Conversions
    - ☑ Implicit
    - ☒ Explicit
  - ☐ Conversions between literals and elementary types: some conversions are not supported
- Units
  - ☐ Ether Units: some units are not supported
  - ☒ Time Units
  - ☐ Special Variables and Functions: some special variables/functions are missing
- Control Structures
  - ☐ Control structures: no support for break, continue, try and catch
  - ☐ Function calls
    - ☑ Internal function calls
    - ☑ External function calls
  - ☒ Creating contracts via new
  - ☐ Assignments
    - ☒ Returning Multiple Values
    - ☑ Complications for Arrays and Structs
  - ☐ Scoping and Declarations: declarations are only allowed at the beginning of a block
  - ☐ Checked or Unchecked Arithmetic: only unchecked is supported
  - ☒ Error Handling
- Contracts
  - ☒ Creating contracts (constructors)
  - ☐ Visibility: only internal and external
  - ☒ Function modifiers
  - ☒ Constant and Immutable state variables
  - ☐ Functions
    - ☑ Parameters and return values
    - ☒ View and Pure functions
    - ☒ Receive function
    - ☑ Fallback function
    - ☒ Function overloading
  - ☒ Inheritance
  - ☒ Interfaces and abstract contracts

  - ☒ Libraries
- ☒ Inline assembly