Isabelle/Solidity

A deep Embedding of Solidity in Isabelle/HOL

Diego Marmsoler and Achim D. Brucker

March 29, 2023

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

Abstract

Smart contracts are automatically executed programs, usually representing legal agreements such as financial transactions. Thus, bugs in smart contracts can lead to large financial losses. For example, an incorrectly initialized contract was the root cause of the Parity Wallet bug that saw $280M worth of Ether destroyed. Ether is the cryptocurrency of the Ethereum blockchain that uses Solidity for expressing smart contracts.

We address this problem by formalizing an executable denotational semantics for Solidity in the interactive theorem prover Isabelle/HOL. This formal semantics builds the foundation of an interactive program verification environment for Solidity programs and allows for inspecting them by (symbolic) execution. We combine the latter with grammar based fuzzing to ensure that our formal semantics complies to the Solidity implementation on the Ethereum Blockchain. Finally, we demonstrate the formal verification of Solidity programs by two examples: constant folding and a simple verified token.

Keywords: Solidity, Denotational Semantics, Isabelle/HOL, Gas

Contents

1 Introduction **7**

2 Preliminaries **9**
2.1 Converting Types to Strings and Back Again (ReadShow) 9
2.2 State Monad with Exceptions (StateMonad) 14

3 Types and Accounts **19**
3.1 Value Types (Valuetypes) 19

4 Stores and Environment **23**
4.1 Storage (Storage) 23
4.2 Environment and State (Environment) 26

5 Expressions and Statements **31**
5.1 Statements (Statements) 31
5.2 The Main Entry Point (Solidity_Main) 49

6 A Solidity Evaluation System **51**
6.1 Towards a Setup for Symbolic Evaluation of Solidity (Solidity_Symbex) 51
6.2 Solidty Evaluator and Code Generator Setup (Solidity_Evaluator) 51
6.3 Generating an Exectuable of the Evaluator (Compile_Evaluator) 62

7 Applications **63**
7.1 Constant Folding (Constant_Folding) 63
7.2 Reentrancy (Reentrancy) 67

1 Introduction

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

For example, the Ethereum blockchain [6] is a feature-rich distributed computing platform that provides not only a cryptocurrency, called *Ether*: Ethereum also provides an immutable distributed data structure (the *blockchain*) on which distributed programs, called *smart contracts*, can be executed. Essentially, smart contracts are automatically executed programs, usually representing a legal agreement, e.g., financial transactions. To support those applications, Ethereum provides a dedicated account data structure on its blockchain that smart contracts can modify, i.e., transferring Ether between accounts. Thus, bugs in smart contracts can lead to large financial losses. For example, an incorrectly initialized contract was the root cause of the Parity Wallet bug that saw $280M worth of Ether destroyed [5]. This risk of bugs being costly is already a big motivation for using formal verification techniques to minimize this risk. The fact that smart contracts are deployed on the blockchain immutably, i.e., they cannot be updated or removed easily, makes it even more important to "get smart contracts" right, before they are deployed on a blockchain for the very first time.

For implementing smart contracts, Ethereum provides *Solidity* [4], a Turing-complete, statically typed programming language that has been designed to look familiar to people knowing Java, C, or JavaScript. Notably, the type system provides, e.g., numerous integer types of different sizes (e.g., `uint256`) and Solidity also relies on different types of stores. While Solidity is Turing-complete, the execution of Solidity programs is guaranteed to terminate. The reason for this is that executing Solidity operations costs *gas*, a tradable commodity on the Ethereum blockchain. Gas does cost Ether and hence, programmers of smart contracts have an incentive to write highly optimized contracts whose execution consumes as little gas as possible. For example, the size of the integer types used can impact the amount of gas required for executing a contract. This desire for highly optimized contracts can conflict with the desire to write correct contracts.

In this paper, we address the problem of developing smart contracts in Solidity that are correct: we present an executable denotational semantics for Solidity in the interactive theorem prover Isabelle/HOL.

In particular, our semantics supports the following features of Solidity:

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

A more abstract description of the semantics is given in [2] and the conformance testing approach for ensuring that our semantics conforms to the actual implementation is described in [3].

The rest of this document is automatically generated from the formalization in Isabelle/HOL, i.e., all content is checked by Isabelle. The structure follows the theory dependencies (see Figure 1.1).

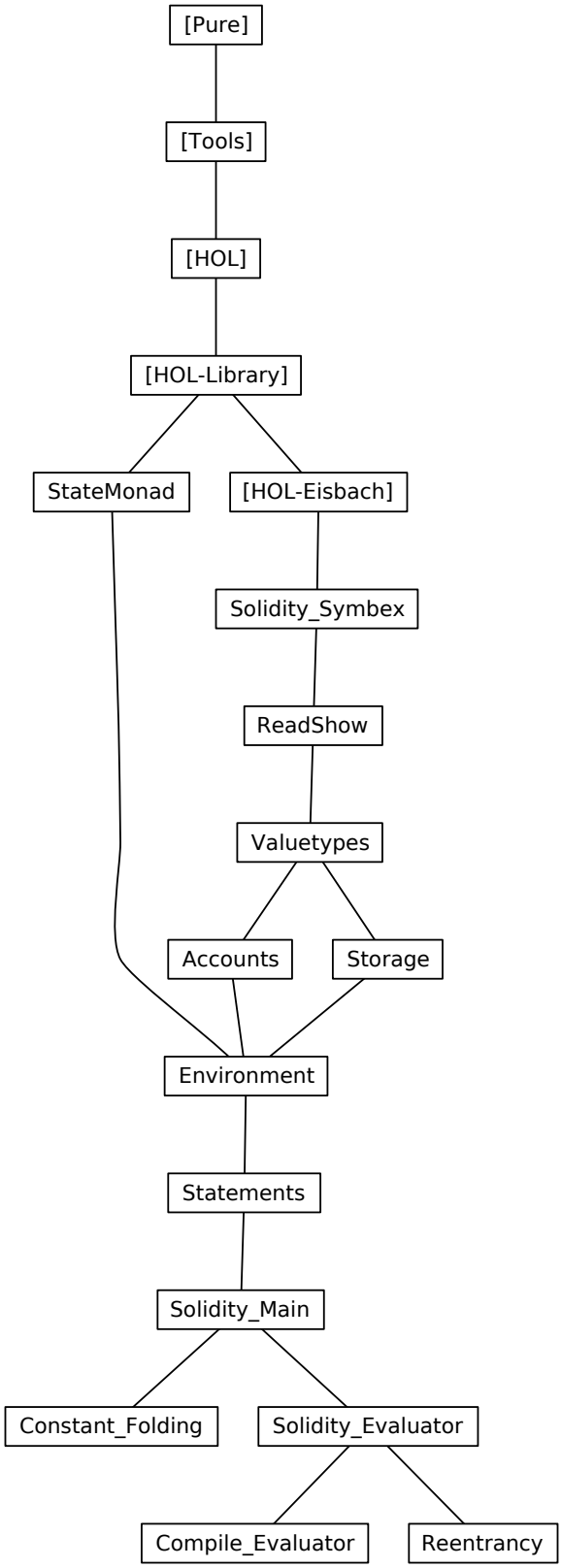

Figure 1.1: The Dependency Graph of the Isabelle Theories.

2 Preliminaries

In this chapter, we discuss auxiliary formalizations and functions that are used in our Solidity semantics but are more generic, i.e., not specific to Solidity. This includes, for example, functions to convert values of basic types to/from strings.

2.1 Converting Types to Strings and Back Again (ReadShow)

theory *ReadShow*
  imports
    Solidity_Symbex
begin

In the following, we formalize a family of projection (and injection) functions for injecting (projecting) basic types (i.e., *nat*, *int*, and *bool* in (out) of the domains of strings. We provide variants for the two string representations of Isabelle/HOL, namely *string* and *String.literal*.

Bool

definition
 ‹$Read_{bool}$ *s = (if s = ''True'' then True else False)*›
definition
 ‹$Show_{bool}$ *b = (if b then ''True'' else ''False'')*›
definition
 ‹*STR_is_bool s =* ($Show_{bool}$ ($Read_{bool}$ *s) = s)*›

declare $Read_{bool}$*_def [solidity_symbex]*
        $Show_{bool}$*_def [solidity_symbex]*

lemma *Show_Read_bool_id:* ‹*STR_is_bool s ⟹* ($Show_{bool}$ ($Read_{bool}$ *s) = s)*›
  ⟨*proof*⟩

lemma *STR_is_bool_split:* ‹*STR_is_bool s ⟹ s = ''False'' ∨ s = ''True''*›
  ⟨*proof*⟩

lemma *Read_Show_bool_id:* ‹$Read_{bool}$ ($Show_{bool}$ *b) = b*›
  ⟨*proof*⟩

definition $ReadL_{bool}$*::*‹*String.literal ⇒ bool*› (‹⌊_⌋›) **where**
 ‹$ReadL_{bool}$ *s = (if s = STR ''True'' then True else False)*›
definition $ShowL_{bool}$*::* ‹*bool ⇒ String.literal*› (‹⌈_⌉›) **where**
 ‹$ShowL_{bool}$ *b = (if b then STR ''True'' else STR ''False'')*›
definition
 ‹*strL_is_bool' s =* ($ShowL_{bool}$ ($ReadL_{bool}$ *s) = s)*›

declare $ReadL_{bool}$*_def [solidity_symbex]*
        $ShowL_{bool}$*_def [solidity_symbex]*

lemma *Show_Read_bool'_id:* ‹*strL_is_bool' s ⟹* ($ShowL_{bool}$ ($ReadL_{bool}$ *s) = s)*›
  ⟨*proof*⟩

lemma *strL_is_bool'_split:* ‹*strL_is_bool' s ⟹ s = STR ''False'' ∨ s = STR ''True''*›
  ⟨*proof*⟩

lemma *Read_Show_bool'_id:* ‹$ReadL_{bool}$ ($ShowL_{bool}$ *b) = b*›
  ⟨*proof*⟩

Natural Numbers

```
definition  nat_of_digit ::  ‹char ⇒ nat› where
  ‹nat_of_digit c =
    (if c = CHR ''0'' then 0
    else if c = CHR ''1'' then 1
    else if c = CHR ''2'' then 2
    else if c = CHR ''3'' then 3
    else if c = CHR ''4'' then 4
    else if c = CHR ''5'' then 5
    else if c = CHR ''6'' then 6
    else if c = CHR ''7'' then 7
    else if c = CHR ''8'' then 8
    else if c = CHR ''9'' then 9
    else undefined)›

declare nat_of_digit_def [solidity_symbex]

definition  digit_of_nat ::  ‹nat ⇒ char› where
  ‹digit_of_nat x =
    (if x = 0 then CHR ''0''
    else if x = 1 then CHR ''1''
    else if x = 2 then CHR ''2''
    else if x = 3 then CHR ''3''
    else if x = 4 then CHR ''4''
    else if x = 5 then CHR ''5''
    else if x = 6 then CHR ''6''
    else if x = 7 then CHR ''7''
    else if x = 8 then CHR ''8''
    else if x = 9 then CHR ''9''
    else undefined)›

declare digit_of_nat_def [solidity_symbex]

lemma nat_of_digit_digit_of_nat_id:
    ‹x < 10 ⟹ nat_of_digit (digit_of_nat x) = x›
  ⟨proof⟩

lemma img_digit_of_nat:
‹n < 10 ⟹ digit_of_nat n ∈ {CHR ''0'', CHR ''1'', CHR ''2'', CHR ''3'', CHR ''4'',
                             CHR ''5'', CHR ''6'', CHR ''7'', CHR ''8'', CHR ''9''}›
  ⟨proof⟩

lemma digit_of_nat_nat_of_digit_id:
    ‹c ∈ {CHR ''0'', CHR ''1'', CHR ''2'', CHR ''3'', CHR ''4'',
          CHR ''5'', CHR ''6'', CHR ''7'', CHR ''8'', CHR ''9''}
      ⟹ digit_of_nat (nat_of_digit c) = c›
  ⟨proof⟩

definition
  nat_implode :: ‹'a::{numeral,power,zero} list ⇒ 'a› where
 ‹nat_implode n = foldr (+) (map (λ (p,d) ⇒ 10 ^ p * d) (enumerate 0 (rev n))) 0›

declare nat_implode_def [solidity_symbex]

fun nat_explode' :: ‹nat ⇒ nat list› where
   ‹nat_explode' x = (case  x < 10 of True ⇒ [x mod 10]
                                    | _ ⇒ (x mod 10 )#(nat_explode' (x div 10)))›

definition
  nat_explode :: ‹nat ⇒ nat list› where
 ‹nat_explode x = (rev (nat_explode' x))›

declare nat_explode_def [solidity_symbex]
```

lemma *nat_explode'_not_empty:* ‹nat_explode' n ≠ []›
⟨*proof*⟩

lemma *nat_explode_not_empty:* ‹nat_explode n ≠ []›
⟨*proof*⟩

lemma *nat_explode'_ne_suc:* ‹∃ n. nat_explode' (Suc n) ≠ nat_explode' n›
⟨*proof*⟩

lemma *nat_explode'_digit:* ‹hd (nat_explode' n) < 10›
⟨*proof*⟩

lemma *div_ten_less:* ‹n ≠ 0 ⟹ ((n::nat) div 10) < n›
⟨*proof*⟩

lemma *unroll_nat_explode':*
‹¬ n < 10 ⟹ (case n < 10 of True ⇒ [n mod 10] | False ⇒ n mod 10 # nat_explode' (n div 10)) =
(n mod 10 # nat_explode' (n div 10))›
⟨*proof*⟩

lemma *nat_explode_mod_10_ident:* ‹map (λ x. x mod 10) (nat_explode' n) = nat_explode' n›
⟨*proof*⟩

lemma *nat_explode'_digits:*
‹∀d ∈ set (nat_explode' n). d < 10›
⟨*proof*⟩

lemma *nat_explode_digits:*
‹∀d ∈ set (nat_explode n). d < 10›
⟨*proof*⟩

value ‹nat_implode(nat_explode 42) = 42›
value ‹nat_explode (Suc 21)›

lemma *nat_implode_append:*
‹nat_implode (a@[b]) = (1*b + foldr (+) (map (λ(p, y). 10 ^ p * y) (enumerate (Suc 0) (rev a))) 0)›
⟨*proof*⟩

lemma *enumerate_suc:* ‹enumerate (Suc n) l = map (λ (a,b). (a+1::nat,b)) (enumerate n l)›
⟨*proof*⟩

lemma *mult_assoc_aux1:*
‹(λ(p, y). 10 ^ p * y) ∘ (λ(a, y). (Suc a, y)) = (λ(p, y). (10::nat) * (10 ^ p) * y)›
⟨*proof*⟩

lemma *fold_map_transfer:*
‹(foldr (+) (map (λ(x,y). 10 * (f (x,y))) l) (0::nat)) = 10 * (foldr (+) (map (λx. (f x)) l)
(0::nat))›
⟨*proof*⟩

lemma *mult_assoc_aux2:* ‹(λ(p, y). 10 * 10 ^ p * (y::nat)) = (λ(p, y). 10 * (10 ^ p * y))›
⟨*proof*⟩

lemma *nat_implode_explode_id:* ‹nat_implode (nat_explode n) = n›
⟨*proof*⟩

definition
$Read_{nat}$ :: ‹string ⇒ nat› **where**
‹$Read_{nat}$ s = nat_implode (map nat_of_digit s)›

definition
$Show_{nat}$::"nat ⇒ string" **where**

```
 ‹Show_nat n = map digit_of_nat (nat_explode n)›

declare Read_nat_def [solidity_symbex]
        Show_nat_def [solidity_symbex]

definition
  ‹STR_is_nat s = (Show_nat (Read_nat s) = s)›

value ‹Read_nat ''10''›
value ‹Show_nat 10›
value ‹Read_nat (Show_nat (10)) = 10›
value ‹Show_nat (Read_nat (''10'')) = ''10''›

lemma Show_nat_not_neg:
  ‹set (Show_nat n) ⊆{CHR ''0'', CHR ''1'', CHR ''2'', CHR ''3'', CHR ''4'',
                      CHR ''5'', CHR ''6'', CHR ''7'', CHR ''8'', CHR ''9''}›
  ⟨proof⟩

lemma Show_nat_not_empty: ‹(Show_nat n) ≠ []›
  ⟨proof⟩

lemma not_hd: ‹L ≠ [] ⟹ e ∉ set(L) ⟹ hd L ≠ e›
  ⟨proof⟩

lemma Show_nat_not_neg'': ‹hd (Show_nat n) ≠ (CHR ''-'')›
  ⟨proof⟩

lemma Show_Read_nat_id: ‹STR_is_nat s ⟹ (Show_nat (Read_nat s) = s)›
  ⟨proof⟩

lemma bar': ‹∀ d ∈ set l . d < 10 ⟹ map nat_of_digit (map digit_of_nat l) = l›
  ⟨proof⟩

lemma Read_Show_nat_id: ‹Read_nat(Show_nat n) = n›
  ⟨proof⟩

definition
  ReadL_nat :: ‹String.literal ⇒ nat› (‹⌈_⌉›) where
 ‹ReadL_nat = Read_nat ∘ String.explode›

definition
  ShowL_nat::‹nat ⇒ String.literal› (‹⌊_⌋›)where
 ‹ShowL_nat = String.implode ∘ Show_nat›

declare ReadL_nat_def [solidity_symbex]
        ShowL_nat_def [solidity_symbex]


definition
  ‹strL_is_nat' s = (ShowL_nat (ReadL_nat s) = s)›

value ‹⌈STR ''10''⌉::nat›
value ‹ReadL_nat (STR ''10'')›
value ‹⌊10::nat⌋›
value ‹ShowL_nat 10›
value ‹ReadL_nat (ShowL_nat (10)) = 10›
value ‹ShowL_nat (ReadL_nat (STR ''10'')) = STR ''10''›

lemma Show_Read_nat'_id: ‹strL_is_nat' s ⟹ (ShowL_nat (ReadL_nat s) = s)›
  ⟨proof⟩

lemma digits_are_ascii:
  ‹c ∈ {CHR ''0'', CHR ''1'', CHR ''2'', CHR ''3'', CHR ''4'',
        CHR ''5'', CHR ''6'', CHR ''7'', CHR ''8'', CHR ''9''}
```

```
   ⟹ String.ascii_of c = c›
```
⟨*proof*⟩

lemma *Show$_{nat}$_ascii:* ‹map String.ascii_of (Show$_{nat}$ n) = Show$_{nat}$ n›
⟨*proof*⟩

lemma *Read_Show_nat'_id:* ‹ReadL_{nat}(ShowL$_{nat}$ n) = n›
⟨*proof*⟩

Integer

definition
```
  Read_int :: ‹string ⇒ int› where
 ‹Read_int x = (if hd x = (CHR ''-'') then  -(int (Read_nat (tl x))) else int (Read_nat x))›
```

definition
```
  Show_int::‹int ⇒ string› where
 ‹Show_int i = (if i < 0 then (CHR ''-'')#(Show_nat (nat (-i)))
                         else Show_nat (nat i))›
```

definition
```
 ‹STR_is_int s = (Show_int (Read_int s) = s)›
```

declare *Read$_{int}$_def [solidity_symbex]*
Show$_{int}$_def [solidity_symbex]

value ‹Read$_{int}$ (Show$_{int}$ 10) = 10›
value ‹Read$_{int}$ (Show$_{int}$ (-10)) = -10›

value ‹Show$_{int}$ (Read$_{int}$ (''10'')) = ''10''›
value ‹Show$_{int}$ (Read$_{int}$ (''-10'')) = ''-10''›

lemma *Show_Read_id:* ‹STR_is_int s ⟹ (Show$_{int}$ (Read$_{int}$ s) = s)›
⟨*proof*⟩

lemma *Read_Show_id:* ‹Read$_{int}$(Show$_{int}$(x)) = x›
⟨*proof*⟩

lemma *STR_is_int_Show:* ‹STR_is_int (Show$_{int}$ n)›
⟨*proof*⟩

definition
```
  ReadL_int :: ‹String.literal ⇒ int› (‹⌈_⌉›) where
 ‹ReadL_int  = Read_int ∘ String.explode›
```

definition
```
  ShowL_int::‹int ⇒ String.literal› (‹⌊_⌋›) where
 ‹ShowL_int =String.implode ∘ Show_int›
```

definition
```
 ‹strL_is_int' s = (ShowL_int (ReadL_int s) = s)›
```

declare *ReadL$_{int}$_def [solidity_symbex]*
ShowL$_{int}$_def [solidity_symbex]

value ‹ReadL$_{int}$ (ShowL$_{int}$ 10) = 10›
value ‹ReadL$_{int}$ (ShowL$_{int}$ (-10)) = -10›

value ‹ShowL$_{int}$ (ReadL$_{int}$ (STR ''10'')) = STR ''10''›
value ‹ShowL$_{int}$ (ReadL$_{int}$ (STR ''-10'')) = STR ''-10''›

lemma *Show_ReadL_id:* ‹strL_is_int' s ⟹ (ShowL$_{int}$ (ReadL$_{int}$ s) = s)›

⟨*proof*⟩

lemma *Read_ShowL_id:* ‹$ReadL_{int}$ ($ShowL_{int}$ x) = x›
⟨*proof*⟩

lemma *STR_is_int_ShowL:* ‹strL_is_int' ($ShowL_{int}$ n)›
⟨*proof*⟩

lemma *String_Cancel:* "a + (c::String.literal) = b + c ⟹ a = b"
⟨*proof*⟩

end
theory *StateMonad*
imports *Main "HOL-Library.Monad_Syntax"*
begin

2.2 State Monad with Exceptions (StateMonad)

datatype *('n, 'e) result = Normal 'n | Exception 'e*

type_synonym *('a, 'e, 's) state_monad = "'s ⇒ ('a × 's, 'e) result"*

lemma *result_cases[cases type: result]:*
fixes *x :: "('a × 's, 'e) result"*
obtains *(n) a s* **where** *"x = Normal (a, s)"*
| *(e) e* **where** *"x = Exception e"*
⟨*proof*⟩

2.2.1 Fundamental Definitions

fun
return :: "'a ⇒ ('a, 'e, 's) state_monad" **where**
"return a s = Normal (a, s)"

fun
throw :: "'e ⇒ ('a, 'e, 's) state_monad" **where**
"throw e s = Exception e"

fun
bind :: "('a, 'e, 's) state_monad ⇒ ('a ⇒ ('b, 'e, 's) state_monad) ⇒
('b, 'e, 's) state_monad" (**infixl** *">>=" 60*)
where
"bind f g s = (case f s of
Normal (a, s') ⇒ g a s'
| Exception e ⇒ Exception e)"

adhoc_overloading *Monad_Syntax.bind bind*

lemma *throw_left[simp]: "throw x ⤜ y = throw x"* ⟨*proof*⟩

2.2.2 The Monad Laws

return is absorbed at the left of a *(⤜)*, applying the return value directly:

lemma *return_bind [simp]: "(return x ⤜ f) = f x"*
⟨*proof*⟩

return is absorbed on the right of a *(⤜)*

lemma *bind_return [simp]: "(m ⤜ return) = m"*
⟨*proof*⟩

(⤜) is associative

lemma *bind_assoc:*
fixes *m :: "('a,'e,'s) state_monad"*
fixes *f :: "'a ⇒ ('b,'e,'s) state_monad"*

fixes *g :: "'b ⇒ ('c,'e,'s) state_monad"*
shows *"(m ≫= f) ≫= g = m ≫= (λx. f x≫= g)"*
⟨*proof*⟩

2.2.3 Basic Conguruence Rules

lemma *monad_cong[fundef_cong]:*
fixes *m1 m2 m3 m4*
assumes *"m1 s = m2 s"*
and *"⋀v s'. m2 s = Normal (v, s') ⟹ m3 v s' = m4 v s'"*
shows *"(bind m1 m3) s = (bind m2 m4) s"*
⟨*proof*⟩

lemma *bind_case_nat_cong [fundef_cong]:*
assumes *"x = x'"* **and** *"⋀a. x = Suc a ⟹ f a h = f' a h"*
shows *"(case x of Suc a ⇒ f a | 0 ⇒ g) h = (case x' of Suc a ⇒ f' a | 0 ⇒ g) h"*
⟨*proof*⟩

lemma *if_cong[fundef_cong]:*
assumes *"b = b'"*
and *"b' ⟹ m1 s = m1' s"*
and *"¬ b' ⟹ m2 s = m2' s"*
shows *"(if b then m1 else m2) s = (if b' then m1' else m2') s"*
⟨*proof*⟩

lemma *bind_case_pair_cong [fundef_cong]:*
assumes *"x = x'"* **and** *"⋀a b. x = (a,b) ⟹ f a b s = f' a b s"*
shows *"(case x of (a,b) ⇒ f a b) s = (case x' of (a,b) ⇒ f' a b) s"*
⟨*proof*⟩

lemma *bind_case_let_cong [fundef_cong]:*
assumes *"M = N"*
and *"(⋀x. x = N ⟹ f x s = g x s)"*
shows *"(Let M f) s = (Let N g) s"*
⟨*proof*⟩

lemma *bind_case_some_cong [fundef_cong]:*
assumes *"x = x'"* **and** *"⋀a. x = Some a ⟹ f a s = f' a s"* **and** *"x = None ⟹ g s = g' s"*
shows *"(case x of Some a ⇒ f a | None ⇒ g) s = (case x' of Some a ⇒ f' a | None ⇒ g') s"*
⟨*proof*⟩

2.2.4 Other functions

The basic accessor functions of the state monad. *get* returns the current state as result, does not fail, and does not change the state. *put s* returns unit, changes the current state to *s* and does not fail.

fun *get :: "('s, 'e, 's) state_monad"* **where**
"get s = Normal (s, s)"

fun *put :: "'s ⇒ (unit, 'e, 's) state_monad"* **where**
"put s _ = Normal ((), s)"

Apply a function to the current state and return the result without changing the state.

fun
applyf :: "('s ⇒ 'a) ⇒ ('a, 'e, 's) state_monad" **where**
"applyf f = get ≫= (λs. return (f s))"

Modify the current state using the function passed in.

fun
modify :: "('s ⇒ 's) ⇒ (unit, 'e, 's) state_monad"
where *"modify f = get ≫= (λs::'s. put (f s))"*

Perform a test on the current state, performing the left monad if the result is true or the right monad if the result is false.

```
fun
  condition :: "('s ⇒ bool) ⇒ ('a, 'e, 's) state_monad ⇒ ('a, 'e, 's) state_monad ⇒ ('a, 'e, 's)
state_monad"
where
  "condition P L R s = (if (P s) then (L s) else (R s))"

notation (output)
  condition  ("(condition (_)//  (_)//  (_))" [1000,1000,1000] 1000)

lemma condition_cong[fundef_cong]:
  assumes "b s = b' s"
    and "b' s ⟹ m1 s = m1' s"
    and "⋀s'. s' = s ⟹ ¬ b' s' ⟹ m2 s' = m2' s'"
  shows "(condition b m1 m2) s = (condition b' m1' m2') s"
  ⟨proof⟩

fun
  assert :: "'e ⇒ ('s ⇒ bool) ⇒ ('a, 'e, 's) state_monad ⇒ ('a, 'e, 's) state_monad" where
 "assert x t m = condition t (throw x) m"

notation (output)
  assert  ("(assert (_)//  (_)//  (_))" [1000,1000,1000] 1000)

lemma assert_cong[fundef_cong]:
  assumes "b s = b' s"
    and "¬ b' s ⟹ m s = m' s"
  shows "(assert x b m) s = (assert x b' m') s"
  ⟨proof⟩
```

2.2.5 Some basic examples

```
lemma "do {
        x ← return 1;
        return (2::nat);
        return x
       } =
       return 1 ≫= (λx. return (2::nat) ≫= (λ_. (return x)))" ⟨proof⟩

lemma "do {
        x ← return 1;
          return 2;
          return x
       } = return 1"
  ⟨proof⟩

fun sub1 :: "(unit, nat, nat) state_monad" where
    "sub1 0 = put 0 0"
  | "sub1 (Suc n) = (do {
                     x ← get;
                     put x;
                     sub1
                    }) n"

fun sub2 :: "(unit, nat, nat) state_monad" where
  "sub2 s =
     (do {
        n ← get;
        (case n of
          0 ⇒ put 0
        | Suc n' ⇒ (do {
                  put n';
                  sub2
                }))
     }) s"
```

```
fun sub3 :: "(unit, nat, nat) state_monad" where
  "sub3 s =
     condition (λn. n=0)
       (return ())
       (do {
         n ← get;
         put (n - 1);
         sub3
        }) s"

fun sub4 :: "(unit, nat, nat) state_monad" where
  "sub4 s =
     assert (0) (λn. n=0)
       (do {
         n ← get;
         put (n - 1);
         sub4
        }) s"

fun sub5 :: "(unit, nat, (nat*nat)) state_monad" where
  "sub5 s =
     assert (0) (λn. fst n=0)
       (do {
         (n,m) ← get;
          put (n - 1,m);
          sub5
        }) s"
end
```

3 Types and Accounts

In this chapter, we discuss the basic data types of Solidity and the representations of accounts.

3.1 Value Types (Valuetypes)

```
theory Valuetypes
imports ReadShow
begin

fun iter :: "(int ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ int ⇒ 'b"
where
  "iter f v x = (if x ≤ 0 then v
                 else f (x-1) (iter f v (x-1)))"

fun iter' :: "(int ⇒ 'b ⇒ 'b option) ⇒ 'b ⇒ int ⇒ 'b option"
where
  "iter' f v x = (if x ≤ 0 then Some v
                  else case iter' f v (x-1) of
                         Some v' ⇒ f (x-1) v'
                       | None ⇒ None)"

type_synonym Address = String.literal
type_synonym Location = String.literal
type_synonym Valuetype = String.literal


datatype Types = TSInt nat
               | TUInt nat
               | TBool
               | TAddr


fun createSInt :: "nat ⇒ int ⇒ Valuetype"
where
  "createSInt b v =
    (if v ≥ 0
      then ShowL_int (-(2^(b-1)) + (v+2^(b-1)) mod (2^b))
      else ShowL_int (2^(b-1) - (-v+2^(b-1)-1) mod (2^b) - 1))"

lemma upper_bound:
  fixes b::nat
    and c::int
  assumes "b > 0"
      and "c < 2^(b-1)"
    shows "c + 2^(b-1) < 2^b"
⟨proof⟩

lemma upper_bound2:
  fixes b::nat
      and c::int
    assumes "b > 0"
      and "c < 2^b"
      and "c ≥ 0"
    shows "c - (2^(b-1)) < 2^(b-1)"
⟨proof⟩

lemma upper_bound3:
```

fixes *b::nat*
and *v::int*
defines *"x ≡ - (2 ^ (b - 1)) + (v + 2 ^ (b - 1)) mod 2 ^ b"*
assumes *"b>0"*
shows *"x < 2^(b-1)"*
⟨*proof*⟩

lemma *lower_bound:*
fixes *b::nat*
assumes *"b>0"*
shows *"∀(c::int) ≥ -(2^(b-1)). (-c + 2^(b-1) - 1 < 2^b)"*
⟨*proof*⟩

lemma *lower_bound2:*
fixes *b::nat*
and *v::int*
defines *"x ≡ 2^(b - 1) - (-v+2^(b-1)-1) mod 2^b - 1"*
assumes *"b>0"*
shows *"x ≥ - (2 ^ (b - 1))"*
⟨*proof*⟩

lemma *createSInt_id_g0:*
fixes *b::nat*
and *v::int*
assumes *"v ≥ 0"*
and *"v < 2^(b-1)"*
and *"b > 0"*
shows *"createSInt b v = ShowL$_{int}$ v"*
⟨*proof*⟩

lemma *createSInt_id_l0:*
fixes *b::nat*
and *v::int*
assumes *"v < 0"*
and *"v ≥ -(2^(b-1))"*
and *"b > 0"*
shows *"createSInt b v = ShowL$_{int}$ v"*
⟨*proof*⟩

lemma *createSInt_id:*
fixes *b::nat*
and *v::int*
assumes *"v < 2^(b-1)"*
and *"v ≥ -(2^(b-1))"*
and *"b > 0"*
shows *"createSInt b v = ShowL$_{int}$ v"* ⟨*proof*⟩

fun *createUInt :: "nat ⇒ int ⇒ Valuetype"*
where *"createUInt b v = ShowL$_{int}$ (v mod (2^b))"*

lemma *createUInt_id:*
assumes *"v ≥ 0"*
and *"v < 2^b"*
shows *"createUInt b v = ShowL$_{int}$ v"*
⟨*proof*⟩

fun *createBool :: "bool ⇒ Valuetype"*
where
"createBool b = ShowL$_{bool}$ b"

fun *createAddress :: "Address ⇒ Valuetype"*
where
"createAddress ad = ad"

```
fun convert :: "Types ⇒ Types ⇒ Valuetype ⇒ (Valuetype * Types) option"
where
  "convert (TSInt b1) (TSInt b2) v =
    (if b1 ≤ b2
      then Some (v, TSInt b2)
      else None)"
| "convert (TUInt b1) (TUInt b2) v =
    (if b1 ≤ b2
      then Some (v, TUInt b2)
      else None)"
| "convert (TUInt b1) (TSInt b2) v =
    (if b1 < b2
      then Some (v, TSInt b2)
      else None)"
| "convert TBool TBool v = Some (v, TBool)"
| "convert TAddr TAddr v = Some (v, TAddr)"
| "convert _ _ _ = None"

lemma convert_id[simp]:
  "convert tp tp kv = Some (kv, tp)"
    ⟨proof⟩


fun olift ::
  "(int ⇒ int ⇒ int) ⇒ Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"
where
  "olift op (TSInt b1) (TSInt b2) v1 v2 =
    Some (createSInt (max b1 b2) (op ⌈v1⌉ ⌈v2⌉), TSInt (max b1 b2))"
| "olift op (TUInt b1) (TUInt b2) v1 v2 =
    Some (createUInt (max b1 b2) (op ⌈v1⌉ ⌈v2⌉), TUInt (max b1 b2))"
| "olift op (TSInt b1) (TUInt b2) v1 v2 =
    (if b2 < b1
      then Some (createSInt b1 (op ⌈v1⌉ ⌈v2⌉), TSInt b1)
      else None)"
| "olift op (TUInt b1) (TSInt b2) v1 v2 =
    (if b1 < b2
      then Some (createSInt b2 (op ⌈v1⌉ ⌈v2⌉), TSInt b2)
      else None)"
| "olift _ _ _ _ _ = None"


fun plift ::
  "(int ⇒ int ⇒ bool) ⇒ Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"
where
  "plift op (TSInt b1) (TSInt b2) v1 v2 = Some (createBool (op ⌈v1⌉ ⌈v2⌉), TBool)"
| "plift op (TUInt b1) (TUInt b2) v1 v2 = Some (createBool (op ⌈v1⌉ ⌈v2⌉), TBool)"
| "plift op (TSInt b1) (TUInt b2) v1 v2 =
    (if b2 < b1
      then Some (createBool (op ⌈v1⌉ ⌈v2⌉), TBool)
      else None)"
| "plift op (TUInt b1) (TSInt b2) v1 v2 =
    (if b1 < b2
      then Some (createBool (op ⌈v1⌉ ⌈v2⌉), TBool)
      else None)"
| "plift _ _ _ _ _ = None"


definition add :: "Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"
where
  "add = olift (+)"


definition sub :: "Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"
```

where
  "sub = olift (-)"

definition *equal :: "Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"*
where
  "equal = plift (=)"

definition *less :: "Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"*
where
  "less = plift (<)"

declare *less_def [solidity_symbex]*

definition *leq :: "Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"*
where
  "leq = plift (≤)"

fun *vtand :: "Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"*
where
  "vtand TBool TBool a b =
    (if a = ShowL$_{bool}$ True ∧ b = ShowL$_{bool}$ True then Some (ShowL$_{bool}$ True, TBool)
    else Some (ShowL$_{bool}$ False, TBool))"
| *"vtand _ _ _ _ = None"*

fun *vtor :: "Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"*
where
  "vtor TBool TBool a b =
    (if a = ShowL$_{bool}$ False ∧ b = ShowL$_{bool}$ False
      then Some (ShowL$_{bool}$ False, TBool)
      else Some (ShowL$_{bool}$ True, TBool))"
| *"vtor _ _ _ _ = None"*

fun *ival :: "Types ⇒ Valuetype"*
where
  "ival (TSInt x) = ShowL$_{int}$ 0"
| *"ival (TUInt x) = ShowL$_{int}$ 0"*
| *"ival TBool = ShowL$_{bool}$ False"*
| *"ival TAddr = STR ''0x0000000000000000000000000000000000000000''"*

end

4 Stores and Environment

In this chapter, we focus on a particular aspect of Solidity that is different to most programming languages: the handling of memory in general and, in particular, the different between store and storage.

4.1 Storage (Storage)

```
theory Storage
imports Valuetypes "HOL-Library.Finite_Map"

begin


fun hash :: "Location ⇒ String.literal ⇒ Location"
where "hash loc ix = ix + (STR ''.'' + loc)"
```

4.1.1 General Store

```
record 'v Store =
  mapping :: "(Location,'v) fmap"
  toploc :: nat

fun accessStore :: "Location ⇒ 'v Store ⇒ 'v option"
where "accessStore loc st = fmlookup (mapping st) loc"

definition emptyStore :: "'v Store"
where "emptyStore = (| mapping=fmempty, toploc=0 |)"

declare emptyStore_def [solidity_symbex]

fun allocate :: "'v Store ⇒ Location * ('v Store)"
where "allocate s = (let ntop = Suc(toploc s) in (ShowL_nat ntop, s (|toploc := ntop|)))"

fun updateStore :: "Location ⇒ 'v ⇒ 'v Store ⇒ 'v Store"
where "updateStore loc val s = s (| mapping := fmupd loc val (mapping s)|)"

fun push :: "'v ⇒ 'v Store ⇒ 'v Store"
  where "push val sto = (let s = updateStore (ShowL_nat (toploc sto)) val sto in snd (allocate s))"
```

4.1.2 Stack

```
datatype Stackvalue = KValue Valuetype
                    | KCDptr Location
                    | KMemptr Location
                    | KStoptr Location


type_synonym Stack = "Stackvalue Store"
```

4.1.3 Storage

Definition

```
type_synonym Storagevalue = Valuetype


type_synonym StorageT = "(Location,Storagevalue) fmap"
```

```
datatype STypes = STArray int STypes
                | STMap Types STypes
                | STValue Types
```

Example

```
abbreviation mystorage::StorageT
where "mystorage ≡ (fmap_of_list
  [(STR ''0.0.1'', STR ''True''),
   (STR ''1.0.1'', STR ''False''),
   (STR ''0.1.1'', STR ''True''),
   (STR ''1.1.1'', STR ''False'')])"
```

Access storage

```
fun accessStorage :: "Types ⇒ Location ⇒ StorageT ⇒ Storagevalue"
where
  "accessStorage t loc sto =
    (case fmlookup sto loc of
      Some v ⇒ v
    | None ⇒ ival t)"
```

Copy from storage to storage

```
fun copyRec :: "Location ⇒ Location ⇒ STypes ⇒ StorageT ⇒ StorageT option"
where
  "copyRec loc loc' (STArray x t) sto =
    iter' (λi s'. copyRec (hash loc (ShowL_int i)) (hash loc' (ShowL_int i)) t s') sto x"
| "copyRec loc loc' (STValue t) sto =
    (let e = accessStorage t loc sto in Some (fmupd loc' e sto))"
| "copyRec _ _ (STMap _ _) _ = None"

fun copy :: "Location ⇒ Location ⇒ int ⇒ STypes ⇒ StorageT ⇒ StorageT option"
where
  "copy loc loc' x t sto =
    iter' (λi s'. copyRec (hash loc (ShowL_int i)) (hash loc' (ShowL_int i)) t s') sto x"
```

4.1.4 Memory and Calldata

Definition

```
datatype Memoryvalue =
  MValue Valuetype
  | MPointer Location

type_synonym MemoryT = "Memoryvalue Store"

type_synonym CalldataT = MemoryT

datatype MTypes = MTArray int MTypes
  | MTValue Types
```

Example

```
abbreviation mymemory::MemoryT
  where "mymemory ≡
    (|mapping = fmap_of_list
      [(STR ''1.1.0'', MValue STR ''False''),
       (STR ''0.1.0'', MValue STR ''True''),
       (STR ''1.0'', MPointer STR ''1.0''),
       (STR ''1.0.0'', MValue STR ''False''),
       (STR ''0.0.0'', MValue STR ''True''),
       (STR ''0.0'', MPointer STR ''0.0'')],
     toploc = 1|)"
```

Initialization

Definition

```
fun minitRec :: "Location ⇒ MTypes ⇒ MemoryT ⇒ MemoryT"
where
  "minitRec loc (MTArray x t) = (λmem.
    let m = updateStore loc (MPointer loc) mem
    in iter (λi m' . minitRec (hash loc (ShowL_int i)) t m') m x)"
| "minitRec loc (MTValue t) = updateStore loc (MValue (ival t))"


fun minit :: "int ⇒ MTypes ⇒ MemoryT ⇒ MemoryT"
where
  "minit x t mem =
    (let l = ShowL_nat (toploc mem);
         m = iter (λi m' . minitRec (hash l (ShowL_int i)) t m') mem x
     in snd (allocate m))"
```

Example

```
lemma "minit 2 (MTArray 2 (MTValue TBool)) emptyStore =
⦇mapping = fmap_of_list
  [(STR ''0.0'', MPointer STR ''0.0''), (STR ''0.0.0'', MValue STR ''False''),
   (STR ''1.0.0'', MValue STR ''False''), (STR ''1.0'', MPointer STR ''1.0''),
   (STR ''0.1.0'', MValue STR ''False''), (STR ''1.1.0'', MValue STR ''False'')],
  toploc = 1⦈" ⟨proof⟩
```

Copy from memory to memory

Definition

```
fun cpm2mrec :: "Location ⇒ Location ⇒ MTypes ⇒ MemoryT ⇒ MemoryT ⇒ MemoryT option"
where
  "cpm2mrec l_s l_d (MTArray x t) m_s m_d =
    (case accessStore l_s m_s of
      Some e ⇒
        (case e of
          MPointer l ⇒ (let m = updateStore l_d (MPointer l_d) m_d
              in iter' (λi m'. cpm2mrec (hash l_s (ShowL_int i)) (hash l_d (ShowL_int i)) t m_s m') m x)
        | _ ⇒ None)
    | None ⇒ None)"
| "cpm2mrec l_s l_d (MTValue t) m_s m_d =
    (case accessStore l_s m_s of
      Some e ⇒ (case e of
          MValue v ⇒ Some (updateStore l_d (MValue v) m_d)
        | _ ⇒ None)
    | None ⇒ None)"

fun cpm2m :: "Location ⇒ Location ⇒ int ⇒ MTypes ⇒ MemoryT ⇒ MemoryT ⇒ MemoryT option"
where
  "cpm2m l_s l_d x t m_s m_d = iter' (λi m. cpm2mrec (hash l_s (ShowL_int i)) (hash l_d (ShowL_int i)) t m_s m)
m_d x"
```

Example

```
lemma "cpm2m (STR ''0'') (STR ''0'') 2 (MTArray 2 (MTValue TBool)) mymemory (snd (allocate
emptyStore)) = Some mymemory"
  ⟨proof⟩
```

4.1.5 Copy from storage to memory

Definition

```
fun cps2mrec :: "Location ⇒ Location ⇒ STypes ⇒ StorageT ⇒ MemoryT ⇒ MemoryT option"
where
```

```
  "cps2mrec locs locm (STArray x t) sto mem =
    (let m = updateStore locm (MPointer locm) mem
    in iter' (λi m'. cps2mrec (hash locs (ShowL_int i)) (hash locm (ShowL_int i)) t sto m') m x)"
| "cps2mrec locs locm (STValue t) sto mem =
    (let v = accessStorage t locs sto
    in Some (updateStore locm (MValue v) mem))"
| "cps2mrec _ _ (STMap _ _) _ _ = None"


fun cps2m :: "Location ⇒ Location ⇒ int ⇒ STypes ⇒ StorageT ⇒ MemoryT ⇒ MemoryT option"
where
  "cps2m locs locm x t sto mem =
    iter' (λi m'. cps2mrec (hash locs (ShowL_int i)) (hash locm (ShowL_int i)) t sto m') mem x"
```

Example

```
lemma "cps2m (STR ''1'') (STR ''0'') 2 (STArray 2 (STValue TBool)) mystorage (snd (allocate
emptyStore)) = Some mymemory"
```
⟨*proof*⟩

4.1.6 Copy from memory to storage

Definition

```
fun cpm2srec :: "Location ⇒ Location ⇒ MTypes ⇒ MemoryT ⇒ StorageT ⇒ StorageT option"
where
  "cpm2srec locm locs (MTArray x t) mem sto =
    (case accessStore locm mem of
      Some e ⇒
        (case e of
          MPointer l ⇒ iter' (λi s'. cpm2srec (hash locm (ShowL_int i)) (hash locs (ShowL_int i)) t mem
s') sto x
        | _ ⇒ None)
    | None ⇒ None)"
| "cpm2srec locm locs (MTValue t) mem sto =
    (case accessStore locm mem of
      Some e ⇒ (case e of
          MValue v ⇒ Some (fmupd locs v sto)
        | _ ⇒ None)
    | None ⇒ None)"


fun cpm2s :: "Location ⇒ Location ⇒ int ⇒ MTypes ⇒ MemoryT ⇒ StorageT ⇒ StorageT option"
where
  "cpm2s locm locs x t mem sto =
    iter' (λi s'. cpm2srec (hash locm (ShowL_int i)) (hash locs (ShowL_int i)) t mem s') sto x"
```

Example

```
lemma "cpm2s (STR ''0'') (STR ''1'') 2 (MTArray 2 (MTValue TBool)) mymemory fmempty = Some mystorage"
```
⟨*proof*⟩

end

4.2 Environment and State (Environment)

```
theory Environment
imports Accounts Storage StateMonad
begin
```

4.2.1 Environment

```
datatype Type = Value Types
              | Calldata MTypes
```

```
            | Memory MTypes
            | Storage STypes

datatype Denvalue = Stackloc Location
                  | Storeloc Location

type_synonym Identifier = String.literal

record Environment =
  address :: Address
  sender :: Address
  svalue :: Valuetype
  denvalue :: "(Identifier, Type × Denvalue) fmap"

fun identifiers :: "Environment ⇒ Identifier fset"
  where "identifiers e = fmdom (denvalue e)"

fun emptyEnv :: "Address ⇒ Address ⇒ Valuetype ⇒ Environment"
  where "emptyEnv a s v = (|address = a, sender = s, svalue =v, denvalue = fmempty|)"

definition eempty :: "Environment"
  where "eempty = emptyEnv (STR '''') (STR '''') (STR '''')"

declare eempty_def [solidity_symbex]

fun updateEnv :: "Identifier ⇒ Type ⇒ Denvalue ⇒ Environment ⇒ Environment"
  where "updateEnv i t v e = e (| denvalue := fmupd i (t,v) (denvalue e) |)"

fun updateEnvOption :: "Identifier ⇒ Type ⇒ Denvalue ⇒ Environment ⇒ Environment option"
  where "updateEnvOption i t v e = (case fmlookup (denvalue e) i of
              Some _ ⇒ None
            | None ⇒ Some (updateEnv i t v e))"

lemma updateEnvOption_address: "updateEnvOption i t v e = Some e' ⟹ address e = address e'"
⟨proof⟩

fun updateEnvDup :: "Identifier ⇒ Type ⇒ Denvalue ⇒ Environment ⇒ Environment"
  where "updateEnvDup i t v e = (case fmlookup (denvalue e) i of
              Some _ ⇒ e
            | None ⇒ updateEnv i t v e)"

lemma updateEnvDup_address[simp]: "address (updateEnvDup i t v e) = address e"
  ⟨proof⟩

lemma updateEnvDup_sender[simp]: "sender (updateEnvDup i t v e) = sender e"
  ⟨proof⟩

lemma updateEnvDup_svalue[simp]: "svalue (updateEnvDup i t v e) = svalue e"
  ⟨proof⟩

lemma updateEnvDup_dup:
  assumes "i≠i'" shows "fmlookup (denvalue (updateEnvDup i t v e)) i' = fmlookup (denvalue e) i'"
⟨proof⟩

lemma env_reorder_neq:
  assumes "x≠y"
  shows "updateEnv x t1 v1 (updateEnv y t2 v2 e) = updateEnv y t2 v2 (updateEnv x t1 v1 e)"
⟨proof⟩

lemma uEO_in:
  assumes "i |∈| fmdom (denvalue e)"
  shows "updateEnvOption i t v e = None"
  ⟨proof⟩
```

```
lemma uEO_n_In:
  assumes "¬ i |∈| fmdom (denvalue e)"
  shows "updateEnvOption i t v e = Some (updateEnv i t v e)"
  ⟨proof⟩

fun astack :: "Identifier ⇒ Type ⇒ Stackvalue ⇒ Stack * Environment ⇒ Stack * Environment"
  where "astack i t v (s, e) = (push v s, (updateEnv i t (Stackloc (ShowL_nat (toploc s))) e))"
```

4.2.2 State

```
type_synonym Gas = nat

record State =
  accounts :: Accounts
  stack :: Stack
  memory :: MemoryT
  storage :: "(Address,StorageT) fmap"
  gas :: Gas

datatype Ex = Gas | Err

fun append :: "Identifier ⇒ Type ⇒ Stackvalue
  ⇒ CalldataT ⇒ Environment ⇒ (CalldataT × Environment, Ex, State) state_monad"
where
  "append id0 tp v cd e st =
    (let (k, e') = astack id0 tp v (stack st, e)
    in do {
      modify (λst. st (|stack := k|));
      return (cd, e')
    }) st"
```

4.2.3 Declarations

This function is used to declare a new variable: decl id tp val copy cd mem cd' env st

id is the name of the variable

tp is the type of the variable

val is an optional initialization parameter. If it is None, the types default value is taken.

copy is a flag to indicate whether memory should be copied (from mem parameter) or not (copying is required for example for external method calls).

cd is the original calldata which is used as a source

mem is the original memory which is used as a source

cd' is the new calldata

env is the new environment

st is the new state

```
fun decl :: "Identifier ⇒ Type ⇒ (Stackvalue * Type) option ⇒ bool ⇒ CalldataT ⇒ MemoryT
    ⇒ CalldataT ⇒ Environment ⇒ (CalldataT × Environment, Ex, State) state_monad"
  where

  "decl i (Value t) None _ _ _ c env st = append i (Value t) (KValue (ival t)) c env st"
| "decl i (Value t) (Some (KValue v, Value t')) _ _ _ c env st =
    (case convert t' t v of
      Some (v', t'') ⇒ append i (Value t'') (KValue v') c env
    | None ⇒ throw Err) st"
| "decl _ (Value _) (Some _) _ _ _ _ _ st = throw Err st"
```

```
| "decl i (Calldata (MTArray x t)) (Some (KCDptr p, _)) True cd _ c env st =
    (let l = ShowL_nat (toploc c);
         (_, c') = allocate c
     in (case cpm2m p l x t cd c' of
          Some c'' ⇒ append i (Calldata (MTArray x t)) (KCDptr l) c'' env
        | None ⇒ throw Err)) st"
| "decl i (Calldata (MTArray x t)) (Some (KMemptr p, _)) True _ mem c env st =
    (let l = ShowL_nat (toploc c);
         (_, c') = allocate c
     in (case cpm2m p l x t mem c' of
          Some c'' ⇒ append i (Calldata (MTArray x t)) (KCDptr l) c'' env
        | None ⇒ throw Err)) st"
| "decl i (Calldata _) _ _ _ _ _ _ st = throw Err st"


| "decl i (Memory (MTArray x t)) None _ _ _ c env st =
    (do {
      m ← applyf (λst. memory st);
      modify (λst. st (|memory := minit x t m|));
      append i (Memory (MTArray x t)) (KMemptr (ShowL_nat (toploc m))) c env
    }) st"
| "decl i (Memory (MTArray x t)) (Some (KMemptr p, _)) True _ mem c env st =
    (do {
      m ← (applyf (λst. memory st));
      (case cpm2m p (ShowL_nat (toploc m)) x t mem (snd (allocate m)) of
         Some m' ⇒
          do {
           modify (λst. st (|memory := m'|));
           append i (Memory (MTArray x t)) (KMemptr (ShowL_nat (toploc m))) c env
          }
       | None ⇒ throw Err)
    }) st"
| "decl i (Memory (MTArray x t)) (Some (KMemptr p, _)) False _ _ c env st =
   append i (Memory (MTArray x t)) (KMemptr p) c env st"
| "decl i (Memory (MTArray x t)) (Some (KCDptr p, _)) _ cd _ c env st =
    (do {
      m ← (applyf (λst. memory st));
      (case cpm2m p (ShowL_nat (toploc m)) x t cd (snd (allocate m)) of
        Some m' ⇒
          do {
          modify (λst. st (|memory := m'|));
          append i (Memory (MTArray x t)) (KMemptr (ShowL_nat (toploc m))) c env
          }
     | None ⇒ throw Err)
    }) st"
| "decl i (Memory (MTArray x t)) (Some (KStoptr p, Storage (STArray x' t'))) _ _ _ c env st =
    (do {
      s ← (applyf (λst. storage st));
      (case fmlookup s (address env) of
        Some s' ⇒
        (do {
          m ← (applyf (λst. memory st));
          (case cps2m p (ShowL_nat (toploc m)) x' t' s' (snd (allocate m)) of
              Some m'' ⇒
              do {
                modify (λst. st (|memory := m''|));
              append i (Memory (MTArray x t)) (KMemptr (ShowL_nat (toploc m))) c env
              }
            | None ⇒ throw Err)
        })
      | None ⇒ throw Err)
    }) st"
| "decl _ (Memory (MTArray _ _)) (Some _) _ _ _ _ _ st = throw Err st"
```

```
| "decl _ (Memory (MTValue _)) _ _ _ _ _ _ st = throw Err st"


| "decl _ (Storage (STArray _ _)) None _ _ _ _ _ st = throw Err st"
| "decl i (Storage (STArray x t)) (Some (KStoptr p, _)) _ _ _ c env st =
    append i (Storage (STArray x t)) (KStoptr p) c env st"
| "decl _ (Storage (STArray _ _)) (Some _) _ _ _ _ _ st = throw Err st"
| "decl _ (Storage (STMap _ _)) None _ _ _ _ _ st = throw Err st"
| "decl i (Storage (STMap t t')) (Some (KStoptr p, _)) _ _ _ c env st =
  append i (Storage (STMap t t')) (KStoptr p) c env st"
| "decl _ (Storage (STMap _ _)) (Some _) _ _ _ _ _ st = throw Err st"
| "decl _ (Storage (STValue _)) _ _ _ _ _ _ st = throw Err st"
```

lemma *decl_gas_address:*
assumes *"decl a1 a2 a3 cp cd mem c env st = Normal ((l1', t1'), st1')"*
shows *"gas st1' = gas st ∧ address env = address t1' ∧ sender env = sender t1' ∧ svalue env = svalue t1'"*
⟨*proof*⟩

end

5 Expressions and Statements

In this chapter, we formalize expressions, declarations, and statements. The results up to here form the core of our Solidity semantics.

5.1 Statements (Statements)

theory *Statements*
  imports *Environment StateMonad*
begin

5.1.1 Syntax

Expressions

```
datatype L = Id Identifier
           | Ref Identifier "E list"
and      E = INT nat int
           | UINT nat int
           | ADDRESS String.literal
           | BALANCE E
           | THIS
           | SENDER
           | VALUE
           | TRUE
           | FALSE
           | LVAL L
           | PLUS E E
           | MINUS E E
           | EQUAL E E
           | LESS E E
           | AND E E
           | OR E E
           | NOT E
           | CALL Identifier "E list"
           | ECALL E Identifier "E list" E
```

Statements

```
datatype S = SKIP
           | BLOCK "(Identifier × Type) × (E option)" S
           | ASSIGN L E
           | TRANSFER E E
           | COMP S S
           | ITE E S S
           | WHILE E S
           | INVOKE Identifier "E list"
           | EXTERNAL E Identifier "E list" E
```

abbreviation
  "vbits≡{8,16,24,32,40,48,56,64,72,80,88,96,104,112,120,128,
  136,144,152,160,168,176,184,192,200,208,216,224,232,240,248,256}"

lemma *vbits_max[simp]:*
  assumes *"b1 ∈ vbits"*
    and *"b2 ∈ vbits"*
  shows *"(max b1 b2) ∈ vbits"*
⟨*proof*⟩

lemma *vbits_ge_0: "(x::nat)∈vbits ⟹ x>0"* ⟨*proof*⟩

5.1.2 Contracts

A contract consists of methods or storage variables. A method is a triple consisting of

- A list of formal parameters
- A statement
- An optional return value

datatype *Member = Method "(Identifier × Type) list × S × E option"*
| Var STypes

A procedure environment assigns a contract to an address. A contract consists of

- An assignment of members to identifiers
- An optional fallback statement which is executed after money is beeing transferred to the contract.

https://docs.soliditylang.org/en/v0.8.6/contracts.html#fallback-function

type_synonym *Environment$_P$ = "(Address, (Identifier, Member) fmap × S) fmap"*

definition *init::"(Identifier, Member) fmap ⇒ Identifier ⇒ Environment ⇒ Environment"*
where *"init ct i e = (case fmlookup ct i of*
Some (Var tp) ⇒ updateEnvDup i (Storage tp) (Storeloc i) e
| _ ⇒ e)"

lemma *init_s11[simp]:*
assumes *"fmlookup ct i = Some (Var tp)"*
shows *"init ct i e = updateEnvDup i (Storage tp) (Storeloc i) e"*
⟨*proof*⟩

lemma *init_s12[simp]:*
assumes *"i |∈| fmdom (denvalue e)"*
shows *"init ct i e = e"*
⟨*proof*⟩

lemma *init_s13[simp]:*
assumes *"fmlookup ct i = Some (Var tp)"*
and *"¬ i |∈| fmdom (denvalue e)"*
shows *"init ct i e = updateEnv i (Storage tp) (Storeloc i) e"*
⟨*proof*⟩

lemma *init_s21[simp]:*
assumes *"fmlookup ct i = None"*
shows *"init ct i e = e"*
⟨*proof*⟩

lemma *init_s22[simp]:*
assumes *"fmlookup ct i = Some (Method m)"*
shows *"init ct i e = e"*
⟨*proof*⟩

lemma *init_commte: "comp_fun_commute (init ct)"*
⟨*proof*⟩

lemma *init_address[simp]:*
"address (init ct i e) = address e ∧ sender (init ct i e) = sender e"
⟨*proof*⟩

lemma *init_sender[simp]:*
"sender (init ct i e) = sender e"

⟨*proof*⟩

lemma *init_svalue[simp]:*
"svalue (init ct i e) = svalue e"
⟨*proof*⟩

lemma *ffold_init_ad_same[rule_format]: "∀e'. ffold (init ct) e xs = e' ⟶ address e' = address e ∧ sender e' = sender e ∧ svalue e' = svalue e"*
⟨*proof*⟩

lemma *ffold_init_dom:*
"fmdom (denvalue (ffold (init ct) e xs)) |⊆| fmdom (denvalue e) |∪| xs"
⟨*proof*⟩

lemma *ffold_init_fmap:*
assumes *"fmlookup ct i = Some (Var tp)"*
and *"i |∉| fmdom (denvalue e)"*
shows *"i|∈|xs ⟹ fmlookup (denvalue (ffold (init ct) e xs)) i = Some (Storage tp, Storeloc i)"*
⟨*proof*⟩

The following definition allows for a more fine-grained configuration of the code generator.

definition *ffold_init::"(String.literal, Member) fmap ⇒ Environment ⇒ String.literal fset ⇒ Environment"* **where**
‹ffold_init ct a c = ffold (init ct) a c›
declare *ffold_init_def [simp]*

lemma *ffold_init_code [code]:*
‹ffold_init ct a c = fold (init ct) (remdups (sorted_list_of_set (fset c))) a›
⟨*proof*⟩

lemma *bind_case_stackvalue_cong [fundef_cong]:*
assumes *"x = x'"*
and *"⋀v. x = KValue v ⟹ f v s = f' v s"*
and *"⋀p. x = KCDptr p ⟹ g p s = g' p s"*
and *"⋀p. x = KMemptr p ⟹ h p s = h' p s"*
and *"⋀p. x = KStoptr p ⟹ i p s = i' p s"*
shows *"(case x of KValue v ⇒ f v | KCDptr p ⇒ g p | KMemptr p ⇒ h p | KStoptr p ⇒ i p) s = (case x' of KValue v ⇒ f' v | KCDptr p ⇒ g' p | KMemptr p ⇒ h' p | KStoptr p ⇒ i' p) s"*
⟨*proof*⟩

lemma *bind_case_type_cong [fundef_cong]:*
assumes *"x = x'"*
and *"⋀t. x = Value t ⟹ f t s = f' t s"*
and *"⋀t. x = Calldata t ⟹ g t s = g' t s"*
and *"⋀t. x = Memory t ⟹ h t s = h' t s"*
and *"⋀t. x = Storage t ⟹ i t s = i' t s"*
shows *"(case x of Value t ⇒ f t | Calldata t ⇒ g t | Memory t ⇒ h t | Storage t ⇒ i t) s = (case x' of Value t ⇒ f' t | Calldata t ⇒ g' t | Memory t ⇒ h' t | Storage t ⇒ i' t) s"*
⟨*proof*⟩

lemma *bind_case_denvalue_cong [fundef_cong]:*
assumes *"x = x'"*
and *"⋀a. x = (Stackloc a) ⟹ f a s = f' a s"*
and *"⋀a. x = (Storeloc a) ⟹ g a s = g' a s"*
shows *"(case x of (Stackloc a) ⇒ f a | (Storeloc a) ⇒ g a) s = (case x' of (Stackloc a) ⇒ f' a | (Storeloc a) ⇒ g' a) s"*
⟨*proof*⟩

lemma *bind_case_mtypes_cong [fundef_cong]:*
assumes *"x = x'"*
and *"⋀a t. x = (MTArray a t) ⟹ f a t s = f' a t s"*
and *"⋀p. x = (MTValue p) ⟹ g p s = g' p s"*
shows *"(case x of (MTArray a t) ⇒ f a t | (MTValue p) ⇒ g p) s = (case x' of (MTArray a t) ⇒ f' a t | (MTValue p) ⇒ g' p) s"*

⟨*proof*⟩

lemma *bind_case_stypes_cong [fundef_cong]:*
assumes "x = x'"
and "⋀a t. x = (STArray a t) ⟹ f a t s = f' a t s"
and "⋀a t. x = (STMap a t) ⟹ g a t s = g' a t s"
and "⋀p. x = (STValue p) ⟹ h p s = h' p s"
shows "(case x of (STArray a t) ⇒ f a t | (STMap a t) ⇒ g a t | (STValue p) ⇒ h p) s
= (case x' of (STArray a t) ⇒ f' a t | (STMap a t) ⇒ g' a t | (STValue p) ⇒ h' p) s"
⟨*proof*⟩

lemma *bind_case_types_cong [fundef_cong]:*
assumes "x = x'"
and "⋀a. x = (TSInt a) ⟹ f a s = f' a s"
and "⋀a. x = (TUInt a) ⟹ g a s = g' a s"
and "x = TBool ⟹ h s = h' s"
and "x = TAddr ⟹ i s = i' s"
shows "(case x of (TSInt a) ⇒ f a | (TUInt a) ⇒ g a | TBool ⇒ h | TAddr ⇒ i) s
= (case x' of (TSInt a) ⇒ f' a | (TUInt a) ⇒ g' a | TBool ⇒ h' | TAddr ⇒ i') s"
⟨*proof*⟩

lemma *bind_case_contract_cong [fundef_cong]:*
assumes "x = x'"
and "⋀a. x = Method a ⟹ f a s = f' a s"
and "⋀a. x = Var a ⟹ g a s = g' a s"
shows "(case x of (Method a) ⇒ f a | (Var a) ⇒ g a) s
= (case x' of (Method a) ⇒ f' a | (Var a) ⇒ g' a) s"
⟨*proof*⟩

lemma *bind_case_memoryvalue_cong [fundef_cong]:*
assumes "x = x'"
and "⋀a. x = MValue a ⟹ f a s = f' a s"
and "⋀a. x = MPointer a ⟹ g a s = g' a s"
shows "(case x of (MValue a) ⇒ f a | (MPointer a) ⇒ g a) s
= (case x' of (MValue a) ⇒ f' a | (MPointer a) ⇒ g' a) s"
⟨*proof*⟩

abbreviation *lift ::*
"(E ⇒ Environment$_P$ ⇒ Environment ⇒ CalldataT ⇒ (Stackvalue * Type, Ex, State) state_monad)
⇒ (Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option)
⇒ E ⇒ E ⇒ Environment$_P$ ⇒ Environment ⇒ CalldataT ⇒ (Stackvalue * Type, Ex, State) state_monad"
where
"lift expr f e1 e2 e_p e cd ≡
(do {
kv1 ← expr e1 e_p e cd;
(case kv1 of
(KValue v1, Value t1) ⇒ (do
{
kv2 ← expr e2 e_p e cd;
(case kv2 of
(KValue v2, Value t2) ⇒
(case f t1 t2 v1 v2 of
Some (v, t) ⇒ return (KValue v, Value t)
| None ⇒ throw Err)
| _ ⇒ (throw Err::(Stackvalue * Type, Ex, State) state_monad))
})
| _ ⇒ (throw Err::(Stackvalue * Type, Ex, State) state_monad))
})"

abbreviation *gascheck ::*
"(State ⇒ Gas) ⇒ (unit, Ex, State) state_monad"
where
"gascheck check ≡
do {

```
    g ← (applyf check::(Gas, Ex, State) state_monad);
    (assert Gas (λst. gas st ≤ g) (modify (λst. st (|gas:=gas st - g|))::(unit, Ex, State) state_monad))
  }"
```

5.1.3 Semantics

```
datatype LType = LStackloc Location
               | LMemloc Location
               | LStoreloc Location


locale statement_with_gas =
  fixes costs :: "S⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒ State ⇒ Gas"
    and costs_e :: "E⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒ State ⇒ Gas"
  assumes while_not_zero[termination_simp]: "⋀e e_p cd st ex s0. 0 < (costs (WHILE ex s0) e_p e cd st) "
      and call_not_zero[termination_simp]: "⋀e e_p cd st i ix. 0 < (costs_e (CALL i ix) e_p e cd st)"
      and ecall_not_zero[termination_simp]: "⋀e e_p cd st a i ix val. 0 < (costs_e (ECALL a i ix val) e_p
e cd st)"
      and invoke_not_zero[termination_simp]: "⋀e e_p cd st i xe. 0 < (costs (INVOKE i xe) e_p e cd st)"
      and external_not_zero[termination_simp]: "⋀e e_p cd st ad i xe val. 0 < (costs (EXTERNAL ad i xe
val) e_p e cd st)"
      and transfer_not_zero[termination_simp]: "⋀e e_p cd st ex ad. 0 < (costs (TRANSFER ad ex) e_p e cd
st)"
begin

function msel::"bool ⇒ MTypes ⇒ Location ⇒ E list ⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒
(Location * MTypes, Ex, State) state_monad"
     and ssel::"STypes ⇒ Location ⇒ E list ⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒ (Location
* STypes, Ex, State) state_monad"
     and lexp :: "L ⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒ (LType * Type, Ex, State)
state_monad"
     and expr::"E ⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒ (Stackvalue * Type, Ex, State)
state_monad"
     and load :: "bool ⇒ (Identifier × Type) list ⇒ E list ⇒ Environment_P ⇒ Environment ⇒
CalldataT ⇒ State ⇒ Environment ⇒ CalldataT ⇒ (Environment × CalldataT × State, Ex, State)
state_monad"
     and rexp::"L ⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒ (Stackvalue * Type, Ex, State)
state_monad"
     and stmt :: "S ⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒ (unit, Ex, State) state_monad"
where
  "msel _ _ _ [] _ _ _ st = throw Err st"
| "msel _ (MTValue _) _ _ _ _ _ st = throw Err st"
| "msel _ (MTArray al t) loc [x] e_p env cd st =
    (do {
      kv ← expr x e_p env cd;
      (case kv of
        (KValue v, Value t') ⇒
          (if less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool)
            then return (hash loc v, t)
            else throw Err)
      | _ ⇒ throw Err)
    }) st"

| "msel mm (MTArray al t) loc (x # y # ys) e_p env cd st =
    (do {
      kv ← expr x e_p env cd;
      (case kv of
        (KValue v, Value t') ⇒
          (if less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool)
            then do {
              s ← applyf (λst. if mm then memory st else cd);
              (case accessStore (hash loc v) s of
                Some (MPointer l) ⇒ msel mm t l (y#ys) e_p env cd
              | _ ⇒ throw Err)
```

```
            } else throw Err)
      | _ ⇒ throw Err)
    }) st"
| "ssel tp loc Nil _ _ _ st = return (loc, tp) st"
| "ssel (STValue _) _ (_ # _) _ _ _ st = throw Err st"
| "ssel (STArray al t) loc (x # xs) e_p env cd st =
    (do {
      kv ← expr x e_p env cd;
      (case kv of
        (KValue v, Value t') ⇒
          (if less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool)
            then ssel t (hash loc v) xs e_p env cd
            else throw Err)
      | _ ⇒ throw Err)
    }) st"
| "ssel (STMap _ t) loc (x # xs) e_p env cd st =
    (do {
      kv ← expr x e_p env cd;
      (case kv of
        (KValue v, _) ⇒ ssel t (hash loc v) xs e_p env cd
      | _ ⇒ throw Err)
    }) st"
| "lexp (Id i) _ e _ st =
    (case fmlookup (denvalue e) i of
      Some (tp, (Stackloc l)) ⇒ return (LStackloc l, tp)
    | Some (tp, (Storeloc l)) ⇒ return (LStoreloc l, tp)
    | _ ⇒ throw Err) st"
| "lexp (Ref i r) e_p e cd st =
    (case fmlookup (denvalue e) i of
      Some (tp, Stackloc l) ⇒
        do {
          k ← applyf (λst. accessStore l (stack st));
          (case k of
            Some (KCDptr _) ⇒ throw Err
          | Some (KMemptr l') ⇒
            (case tp of
              Memory t ⇒
                do {
                  (l'', t') ← msel True t l' r e_p e cd;
                  return (LMemloc l'', Memory t')
                }
            | _ ⇒ throw Err)
          | Some (KStoptr l') ⇒
            (case tp of
              Storage t ⇒
                do {
                  (l'', t') ← ssel t l' r e_p e cd;
                  return (LStoreloc l'', Storage t')
                }
            | _ ⇒ throw Err)
          | Some (KValue _) ⇒ throw Err
          | None ⇒ throw Err)
        }
      | Some (tp, Storeloc l) ⇒
          (case tp of
            Storage t ⇒
              do {
                (l', t') ← ssel t l r e_p e cd;
                return (LStoreloc l', Storage t')
              }
          | _ ⇒ throw Err)
      | None ⇒ throw Err) st"
| "expr (E.INT b x) e_p e cd st =
    (do {
```

```
      gascheck (costs_e (E.INT b x) e_p e cd);
      (if (b ∈ vbits)
        then (return (KValue (createSInt b x), Value (TSInt b)))
        else (throw Err))
    }) st"
| "expr (UINT b x) e_p e cd st =
    (do {
      gascheck (costs_e (UINT b x) e_p e cd);
      (if (b ∈ vbits)
        then (return (KValue (createUInt b x), Value (TUInt b)))
        else (throw Err))
  }) st"
| "expr (ADDRESS ad) e_p e cd st =
    (do {
      gascheck (costs_e (ADDRESS ad) e_p e cd);
      return (KValue ad, Value TAddr)
    }) st"
| "expr (BALANCE ad) e_p e cd st =
    (do {
      gascheck (costs_e (BALANCE ad) e_p e cd);
      kv ← expr ad e_p e cd;
      (case kv of
        (KValue adv, Value TAddr) ⇒
          return (KValue (accessBalance (accounts st) adv), Value (TUInt 256))
      | _ ⇒ throw Err)
    }) st"
| "expr THIS e_p e cd st =
    (do {
      gascheck (costs_e THIS e_p e cd);
      return (KValue (address e), Value TAddr)
    }) st"
| "expr SENDER e_p e cd st =
    (do {
      gascheck (costs_e SENDER e_p e cd);
      return (KValue (sender e), Value TAddr)
    }) st"
| "expr VALUE e_p e cd st =
    (do {
      gascheck (costs_e VALUE e_p e cd);
      return (KValue (svalue e), Value (TUInt 256))
    }) st"
| "expr TRUE e_p e cd st =
    (do {
      gascheck (costs_e TRUE e_p e cd);
      return (KValue (ShowL_bool True), Value TBool)
    }) st"
| "expr FALSE e_p e cd st =
    (do {
      gascheck (costs_e FALSE e_p e cd);
      return (KValue (ShowL_bool False), Value TBool)
    }) st"
| "expr (NOT x) e_p e cd st =
    (do {
      gascheck (costs_e (NOT x) e_p e cd);
      kv ← expr x e_p e cd;
      (case kv of
        (KValue v, Value t) ⇒
          (if v = ShowL_bool True
            then expr FALSE e_p e cd
            else (if v = ShowL_bool False
              then expr TRUE e_p e cd
              else throw Err))
      | _ ⇒ throw Err)
    }) st"
```

```
| "expr (PLUS e1 e2) e_p e cd st = (gascheck (costs_e (PLUS e1 e2) e_p e cd) >>= (λ_. lift expr add e1 e2
e_p e cd)) st"
| "expr (MINUS e1 e2) e_p e cd st = (gascheck (costs_e (MINUS e1 e2) e_p e cd) >>= (λ_. lift expr sub e1
e2 e_p e cd)) st"
| "expr (LESS e1 e2) e_p e cd st = (gascheck (costs_e (LESS e1 e2) e_p e cd) >>= (λ_. lift expr less e1 e2
e_p e cd)) st"
| "expr (EQUAL e1 e2) e_p e cd st = (gascheck (costs_e (EQUAL e1 e2) e_p e cd) >>= (λ_. lift expr equal e1
e2 e_p e cd)) st"
| "expr (AND e1 e2) e_p e cd st = (gascheck (costs_e (AND e1 e2) e_p e cd) >>= (λ_. lift expr vtand e1 e2
e_p e cd)) st"
| "expr (OR e1 e2) e_p e cd st = (gascheck (costs_e (OR e1 e2) e_p e cd) >>= (λ_. lift expr vtor e1 e2 e_p
e cd)) st"
| "expr (LVAL i) e_p e cd st =
    (do {
      gascheck (costs_e (LVAL i) e_p e cd);
      rexp i e_p e cd
    }) st"

| "expr (CALL i xe) e_p e cd st =
    (do {
      gascheck (costs_e (CALL i xe) e_p e cd);
      (case fmlookup e_p (address e) of
         Some (ct, _) ⇒
           (case fmlookup ct i of
             Some (Method (fp, f, Some x)) ⇒
               let e' = ffold_init ct (emptyEnv (address e) (sender e) (svalue e)) (fmdom ct)
               in (do {
                 st' ← applyf (λst. st(|stack:=emptyStore|));
                 (e'', cd', st'') ← load False fp xe e_p e' emptyStore st' e cd;
                 st''' ← get;
                 put st'';
                 stmt f e_p e'' cd';
                 rv ← expr x e_p e'' cd';
                 modify (λst. st(|stack:=stack st''', memory := memory st'''|));
                 return rv
               })
           | _ ⇒ throw Err)
       | None ⇒ throw Err)
    }) st"
| "expr (ECALL ad i xe val) e_p e cd st =
    (do {
      gascheck (costs_e (ECALL ad i xe val) e_p e cd);
      kad ← expr ad e_p e cd;
      (case kad of
        (KValue adv, Value TAddr) ⇒
        (case fmlookup e_p adv of
           Some (ct, _) ⇒
             (case fmlookup ct i of
               Some (Method (fp, f, Some x)) ⇒
               (do {
                 kv ← expr val e_p e cd;
                 (case kv of
                   (KValue v, Value t) ⇒
                     let e' = ffold_init ct (emptyEnv adv (address e) v) (fmdom ct)
                     in (do {
                       st' ← applyf (λst. st(|stack:=emptyStore, memory:=emptyStore|));
                       (e'', cd', st'') ← load True fp xe e_p e' emptyStore st' e cd;
                       st''' ← get;
                       (case transfer (address e) adv v (accounts st'') of
                         Some acc ⇒
                           do {
                             put (st''(|accounts := acc|));
                             stmt f e_p e'' cd';
                             rv ← expr x e_p e'' cd';
```

```
                        modify (λst. st⦇stack:=stack st''', memory := memory st'''⦈);
                        return rv
                      }
                    | None ⇒ throw Err)
                  })
                | _ ⇒ throw Err)
              })
            | _ ⇒ throw Err)
        | None ⇒ throw Err)
      | _ ⇒ throw Err)
    }) st"
| "load cp ((i_p, t_p)#pl) (e#el) e_p e_v' cd' st' e_v cd st =
    (do {
      (v, t) ← expr e e_p e_v cd;
      st'' ← get;
      put st';
      (cd'', e_v'') ← decl i_p t_p (Some (v,t)) cp cd (memory st'') cd' e_v';
      st''' ← get;
      put st'';
      load cp pl el e_p e_v'' cd'' st''' e_v cd
    }) st"
| "load _ [] (_#_) _ _ _ _ _ _ st = throw Err st"
| "load _ (_#_) [] _ _ _ _ _ _ st = throw Err st"
| "load _ [] [] _ e_v' cd' st' e_v cd st = return (e_v', cd', st') st"

| "rexp (Id i) e_p e cd st =
    (case fmlookup (denvalue e) i of
      Some (tp, Stackloc l) ⇒
        do {
          s ← applyf (λst. accessStore l (stack st));
          (case s of
            Some (KValue v) ⇒ return (KValue v, tp)
          | Some (KCDptr p) ⇒ return (KCDptr p, tp)
          | Some (KMemptr p) ⇒ return (KMemptr p, tp)
          | Some (KStoptr p) ⇒ return (KStoptr p, tp)
          | _ ⇒ throw Err)
        }
    | Some (Storage (STValue t), Storeloc l) ⇒
      do {
        so ← applyf (λst. fmlookup (storage st) (address e));
        (case so of
          Some s ⇒ return (KValue (accessStorage t l s), Value t)
        | None ⇒ throw Err)
      }
    | Some (Storage (STArray x t), Storeloc l) ⇒ return (KStoptr l, Storage (STArray x t))
    | _ ⇒ throw Err) st"
| "rexp (Ref i r) e_p e cd st =
    (case fmlookup (denvalue e) i of
      Some (tp, (Stackloc l)) ⇒
        do {
          kv ← applyf (λst. accessStore l (stack st));
          (case kv of
            Some (KCDptr l') ⇒
              (case tp of
                Calldata t ⇒
                  do {
                    (l'', t') ← msel False t l' r e_p e cd;
                    (case t' of
                      MTValue t'' ⇒
                        (case accessStore l'' cd of
                          Some (MValue v) ⇒ return (KValue v, Value t'')
                        | _ ⇒ throw Err)
                    | MTArray x t'' ⇒
                        (case accessStore l'' cd of
```

```
                    Some (MPointer p) ⇒ return (KCDptr p, Calldata (MTArray x t''))
                  | _ ⇒ throw Err))
              }
          | _ ⇒ throw Err)
      | Some (KMemptr l') ⇒
          (case tp of
            Memory t ⇒
            do {
              (l'', t') ← msel True t l' r e_p e cd;
              (case t' of
                MTValue t'' ⇒
                do {
                  mv ← applyf (λst. accessStore l'' (memory st));
                  (case mv of
                    Some (MValue v) ⇒ return (KValue v, Value t'')
                  | _ ⇒ throw Err)
                }
              | MTArray x t'' ⇒
                do {
                  mv ← applyf (λst. accessStore l'' (memory st));
                  (case mv of
                    Some (MPointer p) ⇒ return (KMemptr p, Memory (MTArray x t''))
                  | _ ⇒ throw Err)
                }
              )
            }
          | _ ⇒ throw Err)
      | Some (KStoptr l') ⇒
          (case tp of
            Storage t ⇒
            do {
              (l'', t') ← ssel t l' r e_p e cd;
              (case t' of
                STValue t'' ⇒
                  do {
                    so ← applyf (λst. fmlookup (storage st) (address e));
                    (case so of
                      Some s ⇒ return (KValue (accessStorage t'' l'' s), Value t'')
                    | None ⇒ throw Err)
                  }
              | STArray _ _ ⇒ return (KStoptr l'', Storage t')
              | STMap _ _ ⇒ return (KStoptr l'', Storage t'))
            }
          | _ ⇒ throw Err)
      | _ ⇒ throw Err)
    }
  | Some (tp, (Storeloc l)) ⇒
      (case tp of
        Storage t ⇒
        do {
          (l', t') ← ssel t l r e_p e cd;
          (case t' of
            STValue t'' ⇒
              do {
                so ← applyf (λst. fmlookup (storage st) (address e));
                (case so of
                  Some s ⇒ return (KValue (accessStorage t'' l' s), Value t'')
                | None ⇒ throw Err)
              }
          | STArray _ _ ⇒ return (KStoptr l', Storage t')
          | STMap _ _ ⇒ return (KStoptr l', Storage t'))
        }
      | _ ⇒ throw Err)
  | None ⇒ throw Err) st"
```

```
| "stmt SKIP e_p e cd st = gascheck (costs SKIP e_p e cd) st"
| "stmt (ASSIGN lv ex) e_p env cd st =
    (do {
      gascheck (costs (ASSIGN lv ex) e_p env cd);
      re ← expr ex e_p env cd;
      (case re of
        (KValue v, Value t) ⇒
          do {
            rl ← lexp lv e_p env cd;
            (case rl of
              (LStackloc l, Value t') ⇒
                (case convert t t' v of
                   Some (v', _) ⇒ modify (λst. st (|stack := updateStore l (KValue v') (stack st)|))
                 | None ⇒ throw Err)
            | (LStoreloc l, Storage (STValue t')) ⇒
                (case convert t t' v of
                  Some (v', _) ⇒
                    do {
                      so ← applyf (λst. fmlookup (storage st) (address env));
                      (case so of
                        Some s ⇒ modify (λst. st(|storage := fmupd (address env) (fmupd l v' s)
(storage st)|))
                      | None ⇒ throw Err)
                    }
                | None ⇒ throw Err)
            | (LMemloc l, Memory (MTValue t')) ⇒
                (case convert t t' v of
                  Some (v', _) ⇒ modify (λst. st(|memory := updateStore l (MValue v') (memory st)|))
                | None ⇒ throw Err)
            | _ ⇒ throw Err)
          }
        | (KCDptr p, Calldata (MTArray x t)) ⇒
          do {
            rl ← lexp lv e_p env cd;
            (case rl of
              (LStackloc l, Memory _) ⇒ modify (λst. st (|stack := updateStore l (KCDptr p) (stack
st)|))
            | (LStackloc l, Storage _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  (case sv of
                    Some (KStoptr p') ⇒
                      do {
                        so ← applyf (λst. fmlookup (storage st) (address env));
                        (case so of
                          Some s ⇒
                            (case cpm2s p p' x t cd s of
                              Some s' ⇒ modify (λst. st (|storage := fmupd (address env) s' (storage
st)|))
                            | None ⇒ throw Err)
                        | None ⇒ throw Err)
                      }
                  | _ ⇒ throw Err)
                }
             | (LStoreloc l, _) ⇒
                do {
                  so ← applyf (λst. fmlookup (storage st) (address env));
                  (case so of
                    Some s ⇒
                      (case cpm2s p l x t cd s of
                         Some s' ⇒ modify (λst. st (|storage := fmupd (address env) s' (storage st)|))
                       | None ⇒ throw Err)
                  | None ⇒ throw Err)
                }
```

```
            | (LMemloc l, _) ⇒
              do {
                cs ← applyf (λst. cpm2m p l x t cd (memory st));
                (case cs of
                  Some m ⇒ modify (λst. st (|memory := m|))
                | None ⇒ throw Err)
              }
            | _ ⇒ throw Err)
          }
        | (KMemptr p, Memory (MTArray x t)) ⇒
            do {
              rl ← lexp lv e_p env cd;
              (case rl of
                (LStackloc l, Memory _) ⇒ modify (λst. st(|stack := updateStore l (KMemptr p) (stack
st)|))
              | (LStackloc l, Storage _) ⇒
                  do {
                    sv ← applyf (λst. accessStore l (stack st));
                    (case sv of
                      Some (KStoptr p') ⇒
                      do {
                        so ← applyf (λst. fmlookup (storage st) (address env));
                        (case so of
                          Some s ⇒
                            do {
                              cs ← applyf (λst. cpm2s p p' x t (memory st) s);
                              (case cs of
                              Some s' ⇒ modify (λst. st (|storage := fmupd (address env) s' (storage
st)|))
                              | None ⇒ throw Err)
                              }
                        | None ⇒ throw Err)
                      }
                    | _ ⇒ throw Err)
                  }
              | (LStoreloc l, _) ⇒
                  do {
                    so ← applyf (λst. fmlookup (storage st) (address env));
                    (case so of
                      Some s ⇒
                        do {
                          cs ← applyf (λst. cpm2s p l x t (memory st) s);
                          (case cs of
                            Some s' ⇒ modify (λst. st (|storage := fmupd (address env) s' (storage
st)|))
                          | None ⇒ throw Err)
                        }
                    | None ⇒ throw Err)
                  }
              | (LMemloc l, _) ⇒ modify (λst. st (|memory := updateStore l (MPointer p) (memory st)|))
              | _ ⇒ throw Err)
            }
        | (KStoptr p, Storage (STArray x t)) ⇒
            do {
              rl ← lexp lv e_p env cd;
              (case rl of
                (LStackloc l, Memory _) ⇒
                  do {
                    sv ← applyf (λst. accessStore l (stack st));
                    (case sv of
                      Some (KMemptr p') ⇒
                        do {
                          so ← applyf (λst. fmlookup (storage st) (address env));
                          (case so of
```

```
                        Some s ⇒
                          do {
                            cs ← applyf (λst. cps2m p p' x t s (memory st));
                            (case cs of
                              Some m ⇒ modify (λst. st⦇memory := m⦈)
                            | None ⇒ throw Err)
                          }
                      | None ⇒ throw Err)
                    }
                | _ ⇒ throw Err)
              }
          | (LStackloc l, Storage _) ⇒ modify (λst. st⦇stack := updateStore l (KStoptr p) (stack
st)⦈)
          | (LStoreloc l, _) ⇒
              do {
                so ← applyf (λst. fmlookup (storage st) (address env));
                (case so of
                  Some s ⇒
                    (case copy p l x t s of
                      Some s' ⇒ modify (λst. st ⦇storage := fmupd (address env) s' (storage st)⦈)
                    | None ⇒ throw Err)
                | None ⇒ throw Err)
              }
          | (LMemloc l, _) ⇒
              do {
                so ← applyf (λst. fmlookup (storage st) (address env));
                (case so of
                  Some s ⇒
                    do {
                      cs ← applyf (λst. cps2m p l x t s (memory st));
                      (case cs of
                        Some m ⇒ modify (λst. st⦇memory := m⦈)
                      | None ⇒ throw Err)
                    }
                | None ⇒ throw Err)
              }
          | _ ⇒ throw Err)
        }
      | (KStoptr p, Storage (STMap t t')) ⇒
          do {
            rl ← lexp lv e_p env cd;
            (case rl of
              (LStackloc l, _) ⇒ modify (λst. st⦇stack := updateStore l (KStoptr p) (stack st)⦈)
            | _ ⇒ throw Err)
          }
      | _ ⇒ throw Err)
  }) st"
| "stmt (COMP s1 s2) e_p e cd st =
  (do {
    gascheck (costs (COMP s1 s2) e_p e cd);
    stmt s1 e_p e cd;
    stmt s2 e_p e cd
  }) st"
| "stmt (ITE ex s1 s2) e_p e cd st =
  (do {
    gascheck (costs (ITE ex s1 s2)  e_p e cd);
    v ← expr ex e_p e cd;
    (case v of
      (KValue b, Value TBool) ⇒
          (if b = ShowL_bool True
            then stmt s1 e_p e cd
            else stmt s2 e_p e cd)
    | _ ⇒ throw Err)
  }) st"
```

```
| "stmt (WHILE ex s0) e_p e cd st =
    (do {
      gascheck (costs (WHILE ex s0) e_p e cd);
      v ← expr ex e_p e cd;
      (case v of
        (KValue b, Value TBool) ⇒
          (if b = ShowL_bool True
            then do {
              stmt s0 e_p e cd;
              stmt (WHILE ex s0) e_p e cd
            }
            else return ())
      | _ ⇒ throw Err)
    }) st"
| "stmt (INVOKE i xe) e_p e cd st =
    (do {
      gascheck (costs (INVOKE i xe) e_p e cd);
      (case fmlookup e_p (address e) of
          Some (ct, _) ⇒
            (case fmlookup ct i of
              Some (Method (fp, f, None)) ⇒
                 (let e' = ffold_init ct (emptyEnv (address e) (sender e) (svalue e)) (fmdom ct)
                 in (do {
                    st' ← applyf (λst. (st⦇stack:=emptyStore⦈));
                    (e'', cd', st'') ← load False fp xe e_p e' emptyStore st' e cd;
                    st''' ← get;
                    put st'';
                    stmt f e_p e'' cd';
                    modify (λst. st⦇stack:=stack st''', memory := memory st'''⦈)
                  }))
            | _ ⇒ throw Err)
        | None  ⇒ throw Err)
    }) st"


| "stmt (EXTERNAL ad i xe val) e_p e cd st =
    (do {
      gascheck (costs (EXTERNAL ad i xe val) e_p e cd);
      kad ← expr ad e_p e cd;
      (case kad of
        (KValue adv, Value TAddr) ⇒
          (case fmlookup e_p adv of
            Some (ct, fb) ⇒
              (do {
                kv ← expr val e_p e cd;
                (case kv of
                  (KValue v, Value t) ⇒
                    (case fmlookup ct i of
                      Some (Method (fp, f, None)) ⇒
                      let e' = ffold_init ct (emptyEnv adv (address e) v) (fmdom ct)
                      in (do {
                        st' ← applyf (λst. st⦇stack:=emptyStore, memory:=emptyStore⦈);
                        (e'', cd', st'') ← load True fp xe e_p e' emptyStore st' e cd;
                        st''' ← get;
                        (case transfer (address e) adv v (accounts st'') of
                          Some acc ⇒
                            do {
                              put (st''⦇accounts := acc⦈);
                              stmt f e_p e'' cd';
                              modify (λst. st⦇stack:=stack st''', memory := memory st'''⦈)
                            }
                          | None ⇒ throw Err)
                       })
                    | None ⇒
```

```
                    do {
                      st' ← get;
                      (case transfer (address e) adv v (accounts st') of
                        Some acc ⇒
                          do {
                            st'' ← get;
                            modify (λst. st(|accounts := acc,stack:=emptyStore, memory:=emptyStore|));
                            stmt fb e_p (emptyEnv adv (address e) v) cd;
                            modify (λst. st(|stack:=stack st'', memory := memory st''|))
                          }
                      | None ⇒ throw Err)
                    }
                  | _ ⇒ throw Err)
                | _ ⇒ throw Err)
              })
          | None ⇒ throw Err)
      | _ ⇒ throw Err)
    }) st"
| "stmt (TRANSFER ad ex) e_p e cd st =
    (do {
      gascheck (costs (TRANSFER ad ex) e_p e cd);
      kv ← expr ex e_p e cd;
      (case kv of
        (KValue v, Value t) ⇒
          (do {
            kv' ← expr ad e_p e cd;
            (case kv' of
              (KValue adv, Value TAddr) ⇒
                (do {
                  acs ← applyf accounts;
                  (case transfer (address e) adv v acs of
                    Some acc ⇒ (case fmlookup e_p adv of
                                  Some (ct, f) ⇒
                                    let e' = ffold_init ct (emptyEnv adv (address e) v) (fmdom ct)
                                    in (do {
                                      st' ← get;
                                      modify (λst. (st(|accounts := acc, stack:=emptyStore,
memory:=emptyStore|)));
                                      stmt f e_p e' emptyStore;
                                      modify (λst. st(|stack:=stack st', memory := memory st'|))
                                    })
                                | None ⇒ modify (λst. (st(|accounts := acc|))))
                  | None ⇒ throw Err)
                })
            | _ ⇒ throw Err)
          })
      | _ ⇒ throw Err)
    }) st"
| "stmt (BLOCK ((id0, tp), ex) s) e_p e_v cd st =
    (do {
      gascheck (costs (BLOCK ((id0, tp), ex) s) e_p e_v cd);
      (case ex of
         None ⇒ (do {
           mem ← applyf memory;
           (cd', e') ← decl id0 tp None False cd mem cd e_v;
           stmt s e_p e' cd'
         })
       | Some ex' ⇒ (do {
           (v, t) ← expr ex' e_p e_v cd;
           mem ← applyf memory;
           (cd', e') ← decl id0 tp (Some (v, t)) False cd mem cd e_v;
           stmt s e_p e' cd'
         }))
    }) st"
```

⟨*proof*⟩

5.1.4 Gas Consumption

lemma *lift_gas:*
assumes "lift expr f e1 e2 e_p e cd st = Normal ((v, t), st')"
and "⋀st' v4 t4. expr e1 e_p e cd st = Normal ((v4, t4), st') ⟹ gas st' ≤ gas st"
and "⋀x1 x y xa ya x1a x1b st' v4 t4. expr e1 e_p e cd st = Normal (x, y)
⟹ (xa, ya) = x
⟹ xa = KValue x1a
⟹ ya = Value x1b
⟹ expr e2 e_p e cd y = Normal ((v4, t4), st')
⟹ gas st' ≤ gas y"
shows "gas st' ≤ gas st"
⟨*proof*⟩

lemma *msel_ssel_lexp_expr_load_rexp_stmt_dom_gas:*
"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inl (Inl (c1, t1, l1, xe1, ep1, ev1, cd1, st1)))
⟹ (∀l1' t1' st1'. msel c1 t1 l1 xe1 ep1 ev1 cd1 st1 = Normal ((l1', t1'), st1') ⟶ gas st1' ≤ gas st1)"
"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inl (Inr (Inl (t2, l2, xe2, ep2, ev2, cd2, st2))))
⟹ (∀l2' t2' st2'. ssel t2 l2 xe2 ep2 ev2 cd2 st2 = Normal ((l2', t2'), st2') ⟶ gas st2' ≤ gas st2)"
"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inl (Inr (Inr (l5, ep5, ev5, cd5, st5))))
⟹ (∀l5' t5' st5'. lexp l5 ep5 ev5 cd5 st5 = Normal ((l5', t5'), st5') ⟶ gas st5' ≤ gas st5)"
"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (e4, ep4, ev4, cd4, st4))))
⟹ (∀st4' v4 t4. expr e4 ep4 ev4 cd4 st4 = Normal ((v4, t4), st4') ⟶ gas st4' ≤ gas st4)"
"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inr (lcp, lis, lxs, lep, lev0, lcd0, lst0, lev, lcd, lst))))
⟹ (∀ev cd st st'. load lcp lis lxs lep lev0 lcd0 lst0 lev lcd lst = Normal ((ev, cd, st), st') ⟶ gas st ≤ gas lst0 ∧ gas st' ≤ gas lst ∧ address ev = address lev0)"
"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inr (Inl (l3, ep3, ev3, cd3, st3))))
⟹ (∀l3' t3' st3'. rexp l3 ep3 ev3 cd3 st3 = Normal ((l3', t3'), st3') ⟶ gas st3' ≤ gas st3)"
"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inr (Inr (s6, ep6, ev6, cd6, st6))))
⟹ (∀st6'. stmt s6 ep6 ev6 cd6 st6 = Normal((), st6') ⟶ gas st6' ≤ gas st6)"
⟨*proof*⟩

5.1.5 Termination

lemma *x1:*
assumes "expr x e_p env cd st = Normal (val, s')"
and "msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (x, e_p, env, cd, st))))"
shows "gas s' < gas st ∨ gas s' = gas st"
⟨*proof*⟩

lemma *x2:*
assumes "(if gas st ≤ c then throw Gas st else (get ⤜ (λs. put (s⦇gas := gas s - c⦈))) st) = Normal ((), s')"
and "expr x e_p e cd s' = Normal (val, s'a)"
and "msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (x, e_p, e, cd, s'))))"
shows "gas s'a < gas st ∨ gas s'a = gas st"
⟨*proof*⟩

lemma *x2sub:*
assumes "(if gas st ≤ costs (TRANSFER ad ex) e_p e cd st then throw Gas st
else (get ⤜ (λs. put (s⦇gas := gas s - costs (TRANSFER ad ex) e_p e cd st⦈))) st) = Normal ((), s')" **and**
" expr ex e_p e cd s' = Normal ((KValue vb, Value t), s'a)"
and " msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (ex, e_p, e, cd, s'))))"
and "(⋀ad i xe val e_p e cd st. 0 < costs (EXTERNAL ad i xe val) e_p e cd st)" **and** "gas s'a ≠ gas st"
shows "gas s'a < gas st"
⟨*proof*⟩

lemma *x3:*

assumes *"(if gas st ≤ c then throw Gas st else (get ≫= (λs. put (s⦇gas := gas s - c⦈))) st) = Normal ((), s')"*
and *"s'⦇stack := emptyStore⦈ = va"*
and *"load False ad xe e_p(ffold (init aa) ⦇address = address e, sender = sender e, svalue = svalue e, denvalue = fmempty⦈ (fmdom aa)) emptyStore va e cd s' = Normal ((ag, ah, s'd), vc)"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inr (False, ad, xe, e_p, ffold (init aa) ⦇address = address e, sender = sender e, svalue = svalue e, denvalue = fmempty⦈ (fmdom aa), emptyStore, va, e, cd, s'))))"*
and *"c>0"*
shows *"gas s'd < gas st"*
⟨*proof*⟩

lemma *x4:*
assumes *"(if gas st ≤ c then throw Gas st else (get ≫= (λs. put (s⦇gas := gas s - c⦈))) st) = Normal ((), s')"*
and *"s'⦇stack := emptyStore⦈ = va"*
and *"load False ad xe e_p (ffold (init aa) ⦇address = address e, sender = sender e, svalue = svalue e, denvalue = fmempty⦈ (fmdom aa)) emptyStore va e cd s' = Normal ((ag, ah, s'd), vc)"*
and *"stmt ae e_p ag ah s'd = Normal ((), s'e)"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inr (Inr (ae, e_p, ag, ah, s'd))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inr (False, ad, xe, e_p, ffold (init aa) ⦇address = address e, sender = sender e, svalue = svalue e, denvalue = fmempty⦈ (fmdom aa), emptyStore, va, e, cd, s'))))"*
and *"c>0"*
shows *"gas s'e < gas st"*
⟨*proof*⟩

lemma *x5:*
assumes *"(if gas st ≤ costs (COMP s1 s2) e_p e cd st then throw Gas st else (get ≫= (λs. put (s⦇gas := gas s - costs (COMP s1 s2) e_p e cd st⦈))) st) = Normal ((), s')"*
and *"stmt s1 e_p e cd s' = Normal ((), s'a)"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inr (Inr (s1, e_p, e, cd, s'))))"*
shows *"gas s'a < gas st ∨ gas s'a = gas st"*
⟨*proof*⟩

lemma *x6:*
assumes *"(if gas st ≤ costs (WHILE ex s0) e_p e cd st then throw Gas st else (get ≫= (λs. put (s⦇gas := gas s - costs (WHILE ex s0) e_p e cd st⦈))) st) = Normal ((), s')"*
and *"expr ex e_p e cd s' = Normal (val, s'a)"*
and *"stmt s0 e_p e cd s'a = Normal ((), s'b)"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inr (Inr (s0, e_p, e, cd, s'a))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (ex, e_p, e, cd, s'))))"*
shows *"gas s'b < gas st"*
⟨*proof*⟩

lemma *x7:*
assumes *"(if gas st ≤ c then throw Gas st else (get ≫= (λs. put (s⦇gas := gas s - c⦈))) st) = Normal ((), s')"*
and *"expr ad e_p e cd s' = Normal ((KValue vb, Value TAddr), s'a)"*
and *"expr val e_p e cd s'a = Normal ((KValue vd, Value ta), s'b)"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (val, e_p, e, cd, s'a))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (ad, e_p, e, cd, s'))))"*
and *"c>0"*
shows *"gas s'b < gas st"*
⟨*proof*⟩

lemma *x8:*
assumes *"(if gas st ≤ costs (TRANSFER ad ex) e_p e cd st then throw Gas st else (get ≫= (λs. put (s⦇gas := gas s - costs (TRANSFER ad ex) e_p e cd st⦈))) st) = Normal ((), s')"*
and *"expr ex e_p e cd s' = Normal ((KValue vb, Value t), s'a)"*
and *"expr ad e_p e cd s'a = Normal ((KValue vd, Value TAddr), s'b)"*
and *"s'b⦇accounts := ab, stack := emptyStore, memory := emptyStore⦈ = s'e"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (ad, e_p, e, cd, s'a))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (ex, e_p, e, cd, s'))))"*

shows *"gas s'e < gas st"*
⟨*proof*⟩

lemma *x9:*
assumes *"(if gas st ≤ costs (BLOCK ((id0, tp), Some a) s) e_p e_v cd st then throw Gas st else (get ≫= (λsa. put (sa⦇gas := gas sa - costs (BLOCK ((id0, tp), Some a) s) e_p e_v cd st⦈))) st) = Normal ((), s')"*
and *"expr a e_p e_v cd s' = Normal ((aa, b), s'a)"*
and *"decl id0 tp (Some (aa, b)) False cd vb cd e_v s'a = Normal ((ab, ba), s'c)"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (a, e_p, e_v, cd, s'))))"*
shows *"gas s'c < gas st ∨ gas s'c = gas st"*
⟨*proof*⟩

lemma *x10:*
assumes *"(if gas st ≤ costs (BLOCK ((id0, tp), None) s) e_p e_v cd st then throw Gas st else (get ≫= (λsa. put (sa⦇gas := gas sa - costs (BLOCK ((id0, tp), None) s) e_p e_v cd st⦈))) st) = Normal ((), s')"*
and *"decl id0 tp None False cd va cd e_v s' = Normal ((a, b), s'b)"*
shows *"gas s'b < gas st ∨ gas s'b = gas st"*
⟨*proof*⟩

lemma *x11:*
assumes *"(if gas st ≤ c then throw Gas st else (get ≫= (λs. put (s⦇gas := gas s - c⦈))) st) = Normal ((), s')"*
and *"expr ad e_p e cd s' = Normal ((KValue vb, Value TAddr), s'a)"*
and *"expr val e_p e cd s'a = Normal ((KValue vd, Value ta), s'b)"*
and *"load True af xe e_p (ffold (init ab) ⦇address = vb, sender = address e, svalue = vd, denvalue = fmempty⦈ (fmdom ab)) emptyStore (s'b⦇stack := emptyStore, memory := emptyStore⦈) e cd s'b = Normal ((ak, al, s'g), vh)"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inr (True, af, xe, e_p, ffold (init ab) ⦇address = vb, sender = address e, svalue = vd, denvalue = fmempty⦈ (fmdom ab), emptyStore, s'b⦇stack := emptyStore, memory := emptyStore⦈), e, cd, s'b))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (val, e_p, e, cd, s'a))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (ad, e_p, e, cd, s'))))"*
and *"c>0"*
shows *"gas s'g < gas st"*
⟨*proof*⟩

lemma *x12:*
assumes *"(if gas st ≤ c then throw Gas st else (get ≫= (λs. put (s⦇gas := gas s - c⦈))) st) = Normal ((), s')"*
and *"expr ad e_p e cd s' = Normal ((KValue vb, Value TAddr), s'a)"*
and *"expr val e_p e cd s'a = Normal ((KValue vd, Value ta), s'b)"*
and *"load True af xe e_p (ffold (init ab) ⦇address = vb, sender = address e, svalue = vd, denvalue = fmempty⦈ (fmdom ab)) emptyStore (s'b⦇stack := emptyStore, memory := emptyStore⦈) e cd s'b = Normal ((ak, al, s'g), vh)"*
and *"stmt ag e_p ak al (s'g⦇accounts := ala⦈) = Normal ((), s'h)"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inr (True, af, xe, e_p, ffold (init ab) ⦇address = vb, sender = address e, svalue = vd, denvalue = fmempty⦈ (fmdom ab), emptyStore, s'b⦇stack := emptyStore, memory := emptyStore⦈), e, cd, s'b))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (val, e_p, e, cd, s'a))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inl (Inl (ad, e_p, e, cd, s'))))"*
and *"msel_ssel_lexp_expr_load_rexp_stmt_dom (Inr (Inr (Inr (ag, e_p, ak, al, (s'g⦇accounts := ala⦈))))))"*
and *"c>0"*
shows *"gas s'h < gas st"*
⟨*proof*⟩

termination
⟨*proof*⟩
end

5.1.6 A minimal cost model

fun *costs_min :: "S⇒ Environment$_P$ ⇒ Environment ⇒ CalldataT ⇒ State ⇒ Gas"*

where
"costs_min SKIP e_p e cd st = 0"
| "costs_min (ASSIGN lv ex) e_p e cd st = 0"
| "costs_min (COMP s1 s2) e_p e cd st = 0"
| "costs_min (ITE ex s1 s2) e_p e cd st = 0"
| "costs_min (WHILE ex s0) e_p e cd st = 1"
| "costs_min (TRANSFER ad ex) e_p e cd st = 1"
| "costs_min (BLOCK ((id0, tp), ex) s) e_p e cd st =0"
| "costs_min (INVOKE _ _) e_p e cd st = 1"
| "costs_min (EXTERNAL _ _ _ _) e_p e cd st = 1"

fun *costs_ex :: "E ⇒ Environment$_P$ ⇒ Environment ⇒ CalldataT ⇒ State ⇒ Gas"*
where
"costs_ex (E.INT _ _) e_p e cd st = 0"
| "costs_ex (UINT _ _) e_p e cd st = 0"
| "costs_ex (ADDRESS _) e_p e cd st = 0"
| "costs_ex (BALANCE _) e_p e cd st = 0"
| "costs_ex THIS e_p e cd st = 0"
| "costs_ex SENDER e_p e cd st = 0"
| "costs_ex VALUE e_p e cd st = 0"
| "costs_ex (TRUE) e_p e cd st = 0"
| "costs_ex (FALSE) e_p e cd st = 0"
| "costs_ex (LVAL _) e_p e cd st = 0"
| "costs_ex (PLUS _ _) e_p e cd st = 0"
| "costs_ex (MINUS _ _) e_p e cd st = 0"
| "costs_ex (EQUAL _ _) e_p e cd st = 0"
| "costs_ex (LESS _ _) e_p e cd st = 0"
| "costs_ex (AND _ _) e_p e cd st = 0"
| "costs_ex (OR _ _) e_p e cd st = 0"
| "costs_ex (NOT _) e_p e cd st = 0"
| "costs_ex (CALL _ _) e_p e cd st = 1"
| "costs_ex (ECALL _ _ _ _) e_p e cd st = 1"

global_interpretation *solidity: statement_with_gas costs_min costs_ex*
defines *stmt = "solidity.stmt"*
and *lexp = solidity.lexp*
and *expr = solidity.expr*
and *ssel = solidity.ssel*
and *rexp = solidity.rexp*
and *msel = solidity.msel*
and *load = solidity.load*
⟨*proof*⟩

end

5.2 The Main Entry Point (Solidity_Main)

theory
Solidity_Main
imports
Valuetypes
Storage
Environment
Statements
begin

This theory is the main entry point into the session Solidity, i.e., it serves the same purpose as *Main* for the session HOL.

It is based on Solidity v0.5.16 https://docs.soliditylang.org/en/v0.5.16/index.html

end

6 A Solidity Evaluation System

This chapter discussed a tactic for symbolically executing Solidity statements and expressions as well as provides a configuration for Isabelle's code generator that allows us to generate an efficient implementation of our executable formal semantics in, e.g., Haskell, SML, or Scala. In our test framework, we use Haskell as a target language.

6.1 Towards a Setup for Symbolic Evaluation of Solidity (Solidity_Symbex)

In this chapter, we lay out the foundations for a tactic for executing Solidity statements and expressions symbolically.

```
theory Solidity_Symbex
imports
  Main
  "HOL-Eisbach.Eisbach"
begin

lemma string_literal_cat: "a+b = String.implode ((String.explode a) @ (String.explode b))"
  ⟨proof⟩


lemma string_literal_conv: "(map String.ascii_of y = y) ⟹  (x = String.implode y) = (String.explode
x = y) "
  ⟨proof⟩

lemmas string_literal_opt = Literal.rep_eq zero_literal.rep_eq plus_literal.rep_eq
                            string_literal_cat  string_literal_conv

named_theorems solidity_symbex
method solidity_symbex declares solidity_symbex =
   ((simp add:solidity_symbex cong:unit.case), (simp add:string_literal_opt)?; (code_simp,simp?)+)

declare Let_def [solidity_symbex]
        o_def [solidity_symbex]

end
```

6.2 Solidty Evaluator and Code Generator Setup (Solidity_Evaluator)

```
theory
  Solidity_Evaluator
imports
  Solidity_Main
  "HOL-Library.Code_Target_Numeral"
  "HOL-Library.Sublist"
  "HOL-Library.Finite_Map"
begin
```

6.2.1 Code Generator Setup and Local Tests

Utils

```
definition FAILURE::"String.literal" where "FAILURE = STR ''Failure''"
definition "inta_of_int = int o nat_of_integer"
definition "nat_of_int  = nat_of_integer"
```

```
fun astore :: "Identifier ⇒ Type ⇒ Valuetype ⇒ StorageT * Environment ⇒ StorageT * Environment"
  where "astore i t v (s, e) = (fmupd i v s, (updateEnv i t (Storeloc i) e))"
```

Valuetypes

```
fun dump_Valuetypes::"Types ⇒ Valuetype ⇒ String.literal" where
   "dump_Valuetypes (TSInt _) n = n"
 | "dump_Valuetypes (TUInt _) n = n"
 | "dump_Valuetypes TBool b = (if b = (STR ''True'') then  STR ''true'' else STR ''false'')"
 | "dump_Valuetypes TAddr ad = ad"
```

Generalized Unit Tests

```
lemma "createSInt 8 500 = STR ''-12''"
```
⟨*proof*⟩

```
lemma "STR ''-92134039538802366542421159375273829975''
      = createSInt 128 456484831356494564654654521238948945546546546546546999465"
```
⟨*proof*⟩

```
lemma "STR ''-128'' = createSInt 8 (-128)"
```
⟨*proof*⟩

```
lemma "STR ''244'' = (createUInt 8 500)"
```
⟨*proof*⟩

```
lemma "STR ''220443428915524155977936330922349307608''
      = (createUInt 128 456484831356494564654654521238948945546546546546469994654465465465465168)"
```
⟨*proof*⟩

```
lemma "less (TUInt 144) (TSInt 160) (STR ''5'') (STR ''8'') = Some(STR ''True'', TBool) "
```
⟨*proof*⟩

Load: Accounts

```
fun load_Accounts :: "Accounts ⇒ (Address × Balance) list ⇒ Accounts" where
   "load_Accounts acc []             = acc"
 | "load_Accounts acc ((ad, bal)#as) = load_Accounts (updateBalance ad bal acc) as"

definition dump_Accounts :: "Accounts ⇒ Address list ⇒ String.literal"
where
  "dump_Accounts acc = foldl (λ t a . String.implode (    (String.explode t)
                                                       @ (String.explode a)
                                                       @ ''.balance''
                                                       @ ''==''
                                                       @ String.explode (accessBalance acc a)
                                                       @ ''↩''))
                        (STR '''')"

definition init_Account::"(Address × Balance) list => Accounts" where
  "init_Account = load_Accounts emptyAccount"
```

Load: Store

```
type_synonym Data_Store = "(Location × String.literal) list"

fun show_Store::"'a Store ⇒ 'a fset" where
    "show_Store s = (fmran (mapping s))"
```

Load: Memory

```
datatype Data_Memory = MArray "Data_Memory list"
  | MBool bool
  | MInt int
  | MAddress Address

fun
```

```
 loadRec_Memory :: "Location ⇒ Data_Memory ⇒ MemoryT ⇒ MemoryT" where
"loadRec_Memory loc (MArray dat) mem =
(fst (foldl (λ S d . let (s',x) = S in  (loadRec_Memory (hash loc (ShowL_nat x)) d s', Suc x))
(updateStore loc (MPointer loc) mem,0) dat))"
| "loadRec_Memory loc (MBool b) mem = updateStore loc ((MValue ∘ ShowL_bool) b) mem "
| "loadRec_Memory loc (MInt i) mem = updateStore loc ((MValue ∘ ShowL_int) i) mem "
| "loadRec_Memory loc (MAddress ad) mem = updateStore loc (MValue ad) mem"

definition load_Memory :: "Data_Memory list ⇒ MemoryT ⇒ MemoryT" where
"load_Memory dat mem = (let l     = ShowL_nat (toploc mem);
                          (m, _) = foldl (λ (m',x) d . (loadRec_Memory (((hash l) ∘ ShowL_nat) x) d m',
Suc x)) (mem, 0) dat
    in (snd ∘ allocate) m)"


fun  dumprec_Memory:: "Location ⇒ MTypes ⇒ MemoryT ⇒ String.literal ⇒ String.literal ⇒
String.literal" where
"dumprec_Memory loc tp mem ls str = ( case accessStore loc mem of
    Some (MPointer l) ⇒ ( case tp of
        (MTArray x t) ⇒ iter (λ i str' . dumprec_Memory ((hash l o ShowL_int) i) t mem
                          (ls + (STR ''['') + (ShowL_int i) + (STR '']'')) str') str x
       | _ ⇒ FAILURE)
    | Some (MValue v) ⇒ (case tp of
        MTValue t ⇒ str + ls + (STR ''=='') + dump_Valuetypes t v + (STR ''↩'')
       | _ ⇒ FAILURE)
   | None ⇒ FAILURE)"

definition dump_Memory :: "Location ⇒ int ⇒ MTypes ⇒ MemoryT ⇒ String.literal ⇒String.literal
⇒String.literal" where
"dump_Memory loc x t mem ls str = iter (λi. dumprec_Memory ((hash loc (ShowL_int i))) t mem (ls + STR
''['' + (ShowL_int i + STR '']''))) str x"
```

Storage

```
datatype Data_Storage =
    SArray "Data_Storage list" |
    SMap "(String.literal × Data_Storage) list" |
    SBool bool |
    SInt int |
    SAddress Address

definition splitAt::"nat ⇒ String.literal ⇒ String.literal × String.literal" where
"splitAt n xs = (String.implode(take n (String.explode xs)), String.implode(drop n (String.explode
xs)))"


fun splitOn':: "'a ⇒ 'a list ⇒ 'a list list" where
   "splitOn' x [] acc = [rev acc]"
 | "splitOn' x (y#ys) acc = (if x = y then (rev acc)#(splitOn' x ys [])
                                      else splitOn' x ys (y#acc))"

fun splitOn::"'a  ⇒ 'a list ⇒ 'a list list" where
"splitOn x xs = splitOn' x xs []"

definition isSuffixOf::"String.literal ⇒ String.literal ⇒ bool" where
"isSuffixOf s x = suffix (String.explode s) (String.explode x)"

definition tolist :: "Location ⇒ String.literal list" where
"tolist s = map String.implode (splitOn (CHR ''.'') (String.explode s))"

abbreviation convert :: "Location ⇒ Location"
  where "convert loc ≡ (if loc= STR ''True'' then STR ''true'' else
    if loc=STR ''False'' then STR ''false'' else loc)"
```

```
fun go_Storage :: "Location ⇒ (String.literal ×  STypes) ⇒ (String.literal × STypes)" where
  "go_Storage l (s, STArray _ t) = (s + (STR ''['') + (convert l) + (STR '']''), t)"
| "go_Storage l (s, STMap _ t) = (s + (STR ''['') + (convert l) + (STR '']''), t)"
| "go_Storage l (s, STValue t) = (s + (STR ''['') + (convert l) + (STR '']''), STValue t)"

fun dumpSingle_Storage :: "StorageT ⇒ String.literal ⇒ STypes ⇒ (Location × Location) ⇒
String.literal ⇒ String.literal" where
"dumpSingle_Storage sto id' tp (loc,l) str =
    (case foldr go_Storage (tolist loc) (str + id', tp)  of
        (s, STValue t) ⇒ (case fmlookup sto (loc + l) of
            Some v ⇒ s + (STR ''=='') + dump_Valuetypes t v
          | None ⇒ FAILURE)
        | _ ⇒ FAILURE)"


definition ‹sorted_list_of_set' ≡ map_fun id id (folding_on.F insort [])›

lemma sorted_list_of_fset'_def': ‹sorted_list_of_set' = sorted_list_of_set›
  ⟨proof⟩

lemma sorted_list_of_set_sort_remdups' [code]:
  ‹sorted_list_of_set' (set xs) = sort (remdups xs)›
  ⟨proof⟩

definition  locations_map :: "Location ⇒ (Location, 'v) fmap ⇒ Location list" where
"locations_map loc = (filter (isSuffixOf ((STR ''.'')+loc))) ∘  sorted_list_of_set' ∘ fset ∘ fmdom"

definition  locations :: "Location ⇒ 'v Store ⇒ Location list" where
"locations loc = locations_map loc ∘ mapping"

fun dump_Storage ::  "StorageT ⇒ Location ⇒ String.literal ⇒ STypes ⇒ String.literal ⇒
String.literal"
where
"dump_Storage sto loc id' (STArray _ t) str = foldl
      (λ s l . dumpSingle_Storage sto id' t ((splitAt (length (String.explode l) - length (String.explode
loc) - 1) l)) s
           + (STR ''↩'')) str (locations_map loc sto)"
| "dump_Storage sto loc id' (STMap _ t) str =
   foldl (λ s l . dumpSingle_Storage sto id' t (splitAt (length (String.explode l) - length
(String.explode loc) - 1) l) s + (STR ''↩'')) str
(locations_map loc sto)"
  | "dump_Storage sto loc id' (STValue t) str = (case fmlookup sto loc of
      Some v ⇒ str + id' + (STR ''=='') + dump_Valuetypes t v + (STR ''↩'')
    | _ ⇒ str)"


fun loadRec_Storage :: "Location ⇒ Data_Storage ⇒ StorageT ⇒ StorageT" where
"loadRec_Storage loc (SArray dat) sto = fst (foldl (λ S d . let (s', x) = S in (loadRec_Storage (hash loc
(ShowL_nat x)) d s', Suc x)) (sto,0) dat)"
| "loadRec_Storage loc (SMap dat) sto  = ( foldr (λ (k, v) s'. loadRec_Storage (hash loc k) v s') dat
sto)"
| "loadRec_Storage loc (SBool b) sto = fmupd loc (ShowL_bool b) sto"
| "loadRec_Storage loc (SInt i)  sto = fmupd loc (ShowL_int i) sto"
| "loadRec_Storage loc (SAddress ad) sto = fmupd loc ad sto"
```

Environment

```
datatype Data_Environment =
    Memarr "Data_Memory list" |
    CDarr "Data_Memory list" |
    Stoarr "Data_Storage list"|
    Stomap "(String.literal × Data_Storage) list" |
    Stackbool bool |
    Stobool bool |
```

```
    Stackint int |
    Stoint int |
    Stackaddr Address |
    Stoaddr Address

fun loadsimple_Environment :: "(Stack × CalldataT × MemoryT × StorageT × Environment)
                    ⇒ (Identifier × Type × Data_Environment) ⇒ (Stack × CalldataT × MemoryT ×
StorageT × Environment)"
  where
"loadsimple_Environment (k, c, m, s, e) (id', tp, d) = (case d of
    Stackbool b ⇒
        let (k', e') = astack id' tp (KValue (ShowL_bool b)) (k, e)
        in (k', c, m, s, e')
  | Stobool b ⇒
        let (s', e') = astore id' tp (ShowL_bool b) (s, e)
        in (k, c, m, s', e')
  |  Stackint n ⇒
        let (k', e') = astack id' tp (KValue (ShowL_int n)) (k, e)
        in (k', c, m, s, e')
  |  Stoint n ⇒
        let (s', e') = astore id' tp (ShowL_int n) (s, e)
        in (k, c, m, s', e')
  |  Stackaddr ad ⇒
        let (k', e') = astack id' tp (KValue ad) (k, e)
        in (k', c, m, s, e')
  |  Stoaddr ad ⇒
        let (s', e') = astore id' tp ad (s, e)
        in (k, c, m, s', e')
  |  CDarr a ⇒
        let l = ShowL_nat (toploc c);
            c' = load_Memory a c;
            (k', e') = astack id' tp (KCDptr l) (k, e)
        in (k', c', m, s, e')
  |  Memarr a ⇒
        let l = ShowL_nat (toploc m);
            m' = load_Memory a m;
            (k', e') = astack id' tp (KMemptr l) (k, e)
        in (k', c, m', s, e')
  |  Stoarr a ⇒
        let s' = loadRec_Storage id' (SArray a) s;
            e' = updateEnv id' tp (Storeloc id') e
        in (k, c, m, s', e')
  |  Stomap mp ⇒
        let s' = loadRec_Storage id' (SMap mp) s;
            e' = updateEnv id' tp (Storeloc id') e
        in (k, c, m, s', e')
)"

definition load_Environment::"(Stack × CalldataT × MemoryT × StorageT × Environment) ⇒ (Identifier ×
Type × Data_Environment) list
                    ⇒ (Stack × CalldataT × MemoryT × StorageT × Environment)"
  where
"load_Environment = foldl loadsimple_Environment"

definition getValue_Environment :: "Stack ⇒ CalldataT ⇒ MemoryT ⇒ StorageT ⇒ Environment ⇒
Identifier ⇒ String.literal ⇒ String.literal"
  where
"getValue_Environment k c m s e i txt = (case fmlookup (denvalue e) i of
    Some (tp, Stackloc l) ⇒ (case accessStore l k of
        Some (KValue v) ⇒ (case tp of
            Value t ⇒ (txt + i + (STR ''=='') + dump_Valuetypes t v + (STR ''↩''))
          |  _ ⇒ FAILURE)
      | Some (KCDptr p) ⇒ (case tp of
            Calldata (MTArray x t) ⇒ dump_Memory p x t c i txt
```

```
            | _ ⇒ FAILURE)
      | Some (KMemptr p) ⇒ (case tp of
            Memory (MTArray x t) ⇒ dump_Memory p x t m i txt
            | _ ⇒ FAILURE)
      | Some (KStoptr p) ⇒ (case tp of
            Storage t ⇒ dump_Storage s p i t txt
            | _ ⇒ FAILURE))
   | Some (Storage t, Storeloc l) ⇒ dump_Storage s l i t txt
   | _ ⇒ FAILURE
)"

type_synonym Data_P = "(Address × ((Identifier × Member) list × S))"

definition dump_Environment :: "Stack  ⇒ CalldataT ⇒ MemoryT ⇒ StorageT ⇒ Environment ⇒ Identifier
list ⇒ String.literal"
  where "dump_Environment k c m s e sl = foldr (getValue_Environment k c m s e) sl  (STR '''')"

fun loadProc::"Environment_P ⇒ Data_P ⇒ Environment_P"
  where "loadProc e_P (ad, (xs, fb)) = fmupd ad (fmap_of_list xs, fb) e_P"

fun initStorage::"(Address × Balance) list ⇒ (Address, StorageT) fmap ⇒ (Address, StorageT) fmap"
  where "initStorage [] m = m"
  | "initStorage (x # xs) m = fmupd (fst x) (fmempty) m"
```

6.2.2 Test Setup

```
definition eval::"Gas ⇒ (S ⇒ Environment_P ⇒ Environment ⇒ CalldataT ⇒ (unit, Ex, State)
state_monad)
                ⇒ S ⇒ Address ⇒ Address ⇒ Valuetype ⇒ (Address × Balance) list
                ⇒ Data_P list
                ⇒ (String.literal × Type × Data_Environment) list
          ⇒ String.literal"
  where "eval g stmt_eval stm addr adest aval acc d dat
       = (let (k,c,m,s,e) = load_Environment (emptyStore, emptyStore, emptyStore, fmempty, emptyEnv
addr adest aval) dat;
              e_p        = foldl loadProc fmempty d;
              a          = init_Account acc;
              s'         = fmupd addr s (initStorage acc fmempty);
              z          = (|accounts=a,stack=k,memory=m,storage=s',gas=g|)
          in (
            case (stmt_eval stm e_p e c z) of
             Normal ((), z') ⇒ (dump_Environment (stack z') c (memory z') (the (fmlookup (storage z')
addr)) e (map (λ (a,b,c). a) dat))
                             + (dump_Accounts (accounts z') (map fst acc))
            | Exception Err   ⇒ STR ''Exception''
            | Exception Gas   ⇒ STR ''OutOfGas''))"

value "eval 1
           stmt
           SKIP
           (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
           (STR '''')
           (STR ''0'')
           [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''),(STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'')]
           []
           [(STR ''v1'', (Value TBool, Stackbool True))]"

lemma "eval 1000
           stmt
           SKIP
           (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
           (STR '''')
           (STR ''0'')
```

```
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'')]
            []
            [(STR ''v1'', (Value TBool, Stackbool True))]
     = STR ''v1==true ↩ 089Be5381FcEa58aF334101414c04F993947C733.balance==100 ↩ 115f6e2F70210C14f7DB1AC69737a3CC78
  ⟨proof⟩

value "eval 1000
            stmt
            SKIP
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            (STR '''')
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'')]
            []
            [(STR ''v1'', (Storage (STArray 5 (STValue TBool)), Stoarr [SBool True, SBool False, SBool
True, SBool False, SBool True]))]"


lemma "eval 1000
            stmt
            SKIP
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            (STR '''')
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'')]
            []
            [(STR ''v1'',(Memory (MTArray 5 (MTValue TBool)), Memarr [MBool True, MBool False, MBool
True, MBool False, MBool True]))]
       = STR ''v1[0]==true ↩ v1[1]==false ↩ v1[2]==true ↩ v1[3]==false ↩ v1[4]==true ↩ 089Be5381FcEa58aF3341014
  ⟨proof⟩

lemma "eval 1000
            stmt
            (ITE FALSE (ASSIGN (Id (STR ''x'')) TRUE) (ASSIGN (Id (STR ''y'')) TRUE))
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            (STR '''')
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'')]
            []
            [(STR ''x'', (Value TBool, Stackbool False)), (STR ''y'', (Value TBool, Stackbool False))]
       = STR ''y==true ↩ x==false ↩ 089Be5381FcEa58aF334101414c04F993947C733.balance==100 ↩ 115f6e2F70210C14f7DB1
  ⟨proof⟩

lemma "eval 1000
            stmt
            (BLOCK ((STR ''v2'', Value TBool), None) (ASSIGN (Id (STR ''v1'')) (LVAL (Id (STR
''v2'')))))
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            (STR '''')
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'')]
            []
            [(STR ''v1'', (Value TBool, Stackbool True))]
       = STR ''v1==false ↩ 089Be5381FcEa58aF334101414c04F993947C733.balance==100 ↩ 115f6e2F70210C14f7DB1AC69737a3
  ⟨proof⟩

lemma "eval 1000
            stmt
            (ASSIGN (Id (STR ''a_s120_21_m8'')) (LVAL (Id (STR ''a_s120_21_s8''))))
```

```
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            (STR '''')
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'')]
            []
            [((STR ''a_s120_21_s8''), Storage (STArray 1 (STArray 2 (STValue (TSInt 120)))), Stoarr
[SArray [SInt 347104507864064359095275590289383142, SInt 565831699297331399489670920129618233]]),
             ((STR ''a_s120_21_m8''), Memory (MTArray 1 (MTArray 2 (MTValue (TSInt 120)))), Memarr
[MArray [MInt (290845675805142398428016622247257774), MInt ((-96834026877269277170645294669272226))]])]
= STR ''a_s120_21_m8[0][0]==347104507864064359095275590289383142↩a_s120_21_m8[0][1]==565831699297331399489670920
  ⟨proof⟩

lemma "eval 1000
            stmt
            (ASSIGN (Ref (STR ''a_s8_32_m0'') [UINT 8 1]) (LVAL (Ref (STR ''a_s8_31_s7'') [UINT 8 0])))
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            (STR '''')
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'')]
            []
            [(STR ''a_s8_31_s7'', (Storage (STArray 1 (STArray 3 (STValue (TSInt 8)))), Stoarr [SArray
[SInt ((98)), SInt ((-23)), SInt (36)]])),
             (STR ''a_s8_32_m0'', (Memory (MTArray 2 (MTArray 3 (MTValue (TSInt 8)))), Memarr [MArray
[MInt ((-64)), MInt ((39)), MInt ((-125))], MArray [MInt ((-32)), MInt ((-82)), MInt ((-105))]]))]
       = STR ''a_s8_32_m0[0][0]==-64↩a_s8_32_m0[0][1]==39↩a_s8_32_m0[0][2]==-125↩a_s8_32_m0[1][0]==98↩a_
  ⟨proof⟩


lemma "eval 1000
            stmt
            SKIP
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            (STR '''')
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'')]
            []
            [(STR ''v1'', (Storage (STMap (TUInt 32) (STValue (TUInt 8))), Stomap [(STR ''2129136830'',
SInt (247))]))]
       = STR ''v1[2129136830]==247↩089Be5381FcEa58aF334101414c04F993947C733.balance==100↩115f6e2F70210C14f7DB1
  ⟨proof⟩

value "eval 1000
            stmt
            (INVOKE (STR ''m1'') [])
            (STR ''myaddr'')
            (STR '''')
            (STR ''0'')
            [(STR ''myaddr'', STR ''100'')]
            [
              (STR ''myaddr'',
                ([(STR ''m1'', Method ([], SKIP, None))],
                SKIP))
            ]
            [(STR ''x'', (Value TBool, Stackbool True))]"

lemma "eval 1000
            stmt
            (ASSIGN (Id (STR ''v1'')) (CALL (STR ''m1'') []))
            (STR ''myaddr'')
            (STR '''')
            (STR ''0'')
            [(STR ''myaddr'', STR ''100'')]
            [
```

```
            (STR ''myaddr'',
              ([(STR ''m1'', Method ([], SKIP, Some (UINT 8 5)))],
              SKIP))
          ]
          [(STR ''v1'', (Value (TUInt 8), Stackint 0))]
     = STR ''v1==5↩myaddr.balance==100↩''"
```
⟨*proof*⟩

lemma *"eval 1000*
```
          stmt
          (ASSIGN (Id (STR ''v1'')) (CALL (STR ''m1'') [E.INT 8 3, E.INT 8 4]))
          (STR ''myaddr'')
          (STR '''')
          (STR ''0'')
          [(STR ''myaddr'', STR ''100'')]
          [
            (STR ''myaddr'',
              ([(STR ''m1'', Method ([(STR ''v2'', Value (TSInt 8)), (STR ''v3'', Value (TSInt 8))],
SKIP, Some (PLUS (LVAL (Id (STR ''v2''))) (LVAL (Id (STR ''v3'')))))],
              SKIP))
          ]
          [(STR ''v1'', (Value (TSInt 8), Stackint 0))]
     = STR ''v1==7↩myaddr.balance==100↩''"
```
⟨*proof*⟩

lemma *"eval 1000*
```
          stmt
          (ASSIGN (Id (STR ''v1'')) (ECALL (ADDRESS (STR ''extaddr'')) (STR ''m1'') [E.INT 8 3, E.INT
8 4] (E.UINT 8 0)))
          (STR ''myaddr'')
          (STR '''')
          (STR ''0'')
          [(STR ''myaddr'', STR ''100'')]
          [
            (STR ''extaddr'',
              ([(STR ''m1'', Method ([(STR ''v2'', Value (TSInt 8)), (STR ''v3'', Value (TSInt 8))],
SKIP, Some (PLUS (LVAL (Id (STR ''v2''))) (LVAL (Id (STR ''v3'')))))],
              SKIP))
          ]
          [(STR ''v1'', (Value (TSInt 8), Stackint 0))]
     = STR ''v1==7↩myaddr.balance==100↩''"
```
⟨*proof*⟩

lemma *"eval 1000*
```
          stmt
          (TRANSFER (ADDRESS (STR ''myaddr'')) (UINT 256 10))
          (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
          (STR '''')
          (STR ''0'')
          [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''0x2d5F6f401c770eEAdd68deB348948ed4504c4676'', STR ''100'')]
          [
            (STR ''myaddr'',
              ([], SKIP))
          ]
          []
     = STR ''089Be5381FcEa58aF334101414c04F993947C733.balance==90↩0x2d5F6f401c770eEAdd68deB348948ed4504c4676.bal
```
⟨*proof*⟩

value *"eval 1000*
```
          stmt
          (TRANSFER (ADDRESS (STR ''myaddr'')) (UINT 256 10))
          (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
          (STR '''')
```

```
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''0x2d5F6f401c770eEAdd68deB348948ed4504c4676'', STR ''100'')]
            [
              (STR ''myaddr'',
                ([], SKIP))
            ]
            []"

lemma "eval 1000
            stmt
            (COMP(COMP(((ASSIGN (Id (STR ''x''))  (E.UINT 8 0))))(TRANSFER (ADDRESS (STR ''myaddr''))
(UINT 256 5)))(SKIP))
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            (STR '''')
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100''), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'')]
            [
              (STR ''myaddr'',
                ([], SKIP))
            ]
            [(STR ''x'', (Value (TUInt 8), Stackint 9))]
      = STR ''x==0↩089Be5381FcEa58aF334101414c04F993947C733.balance==95↩115f6e2F70210C14f7DB1AC69737a3CC78435d
  ⟨proof⟩

value "eval 1000
            stmt
            (EXTERNAL (ADDRESS (STR ''Victim'')) (STR ''withdraw'') [] (E.UINT 8 0))
            (STR ''Victim'')
            (STR '''')
            (STR ''0'')
            [(STR ''Victim'', STR ''100''), (STR ''Attacker'', STR ''100'')]
            [
              (STR ''Attacker'',
                [(STR ''withdraw'', Method ([], EXTERNAL (ADDRESS (STR ''Victim'')) (STR ''withdraw'')
[] (E.UINT 8 0), None))],
                SKIP),
              (STR ''Victim'',
                [(STR ''withdraw'', Method ([], EXTERNAL (ADDRESS (STR ''Attacker'')) (STR ''withdraw'')
[] (E.UINT 8 0), None))],
                SKIP)
            ]
            []"

value "eval 1000
            stmt
            (INVOKE (STR ''withdraw'') [])
            (STR ''Victim'')
            (STR '''')
            (STR ''0'')
            [(STR ''Victim'', STR ''100''), (STR ''Attacker'', STR ''100'')]
            [
              (STR ''Victim'',
                [(STR ''withdraw'', Method ([], INVOKE (STR ''withdraw'') [], None))],
                SKIP)
            ]
            []"
```

6.2.3 The Final Code Export

consts $ReadL_S$ :: "String.literal ⇒ S"
consts $ReadL_{acc}$ :: "String.literal ⇒ (String.literal × String.literal) list"
consts $ReadL_{dat}$ :: "String.literal ⇒ (String.literal × Type × $Data_{Environment}$) list"

```
consts ReadL_P :: "String.literal ⇒ Data_P list"

code_printing
   constant ReadL_S   ⇀ (Haskell) "Prelude.read"
 | constant ReadL_acc ⇀ (Haskell) "Prelude.read"
 | constant ReadL_dat ⇀ (Haskell) "Prelude.read"
 | constant ReadL_P   ⇀ (Haskell) "Prelude.read"

fun main_stub :: "String.literal list ⇒ (int × String.literal)"
  where
   "main_stub [credit, stm, saddr, raddr, val, acc, pr, dat]
        = (0, eval (ReadL_nat credit) stmt (ReadL_S stm) saddr raddr val (ReadL_acc acc) (ReadL_P pr)
(ReadL_dat dat))"
 | "main_stub [stm, saddr, raddr, val, acc, pr, dat]
        = (0, eval 1000 stmt (ReadL_S stm) saddr raddr val (ReadL_acc acc) (ReadL_P pr) (ReadL_dat dat))"
 | "main_stub _ = (2,
          STR ''solidity-evaluator [credit] "Statement" "ContractAddress" "OriginAddress" "Value"↩''
        + STR ''  "(Address * Balance) list" "(Address * ((Identifier * Member) list) * Statement)"
"(Variable * Type * Value) list"↩''
        + STR ''↩''')"

generate_file "code/solidity-evaluator/app/Main.hs" = <
module Main where
import System.Environment
import Solidity_Evaluator
import Prelude

main :: IO ()
main = do
  args <- getArgs
  Prelude.putStr(snd $ Solidity_Evaluator.main_stub args)
>

export_generated_files _

export_code eval SKIP main_stub
           in Haskell module_name "Solidity_Evaluator" file_prefix "solidity-evaluator/src"
(string_classes)
```

6.2.4 Demonstrating the Symbolic Execution of Solidity

```
abbreviation P1::S
  where "P1 ≡ COMP (ASSIGN (Id (STR ''sa'')) (LVAL (Id (STR ''ma''))))
                   (ASSIGN (Ref (STR ''sa'') [UINT (8::nat) 0]) TRUE)"
abbreviation myenv::Environment
  where "myenv ≡ updateEnv (STR ''ma'') (Memory (MTArray 1 (MTValue TBool))) (Stackloc (STR ''1''))
                (updateEnv (STR ''sa'') (Storage (STArray 1 (STValue TBool))) (Storeloc (STR ''1''))
                (emptyEnv (STR ''ad'') (STR ''ad'') (STR ''0'')))"

abbreviation mystack::Stack
  where "mystack ≡ updateStore (STR ''1'') (KMemptr (STR ''1'')) emptyStore"

abbreviation mystore::StorageT
  where "mystore ≡ fmempty"

abbreviation mymemory::MemoryT
  where "mymemory ≡ updateStore (STR ''0.1'') (MValue (STR ''false'')) emptyStore"

abbreviation mystorage::StorageT
  where "mystorage ≡ fmupd (STR ''0.1'') (STR ''True'') fmempty"


lemma "( stmt P1 fmempty myenv emptyStore (|accounts=emptyAccount, stack=mystack, memory=mymemory,
storage=fmupd (STR ''ad'') mystorage fmempty, gas=1000|) ) =
```

```
  (Normal ((), (|accounts=emptyAccount, stack=mystack, memory=mymemory, storage=fmupd (STR ''ad'')
mystorage fmempty, gas=1000|) ))"
  ⟨proof⟩
```

end

6.3 Generating an Exectuable of the Evaluator (Compile_Evaluator)

theory
  Compile_Evaluator
  imports
    Solidity_Evaluator
begin

```
compile_generated_files _ (in Solidity_Evaluator) export_files "solidity-evaluator" (executable)
  where <fn dir =>
    let
       val modules_src =
        Generated_Files.get_files theory<Solidity_Evaluator>
        |> filter (fn p => String.isSubstring "src" (Path.implode (#path p)))
        |> map (#path #> Path.implode  #> unsuffix ".hs" #> space_explode "/" #> space_implode "."
                      #> unprefix "code.solidity-evaluator.src.");
       val modules_app =
        Generated_Files.get_files theory<Solidity_Evaluator>
        |> filter (fn p => String.isSubstring "app" (Path.implode (#path p)))
        |> map (#path #> Path.implode  #> unsuffix ".hs" #> space_explode "/" #> space_implode "."
                      #> unprefix "code.solidity-evaluator.app.");
      val _ =
        GHC.new_project dir
          {name = "solidity-evaluator",
           depends =
            [],
           modules = modules_app};
     val _ = writeln (Path.implode dir)
     val res = Generated_Files.execute dir <Build> (String.concat [
                  "echo \"\n  default-extensions:  TypeSynonymInstances, FlexibleInstances\" >>
solidity-evaluator.cabal"
                 ," && rm -rf src"
                 ," && mv code/solidity-evaluator/src src"
                 ," && mv code/solidity-evaluator/app/* src/"
                 ," && isabelle ghc_stack install --local-bin-path . `pwd`"])
   in
     writeln (res)
   end>
```

end

7 Applications

In this chapter, we discuss various applications of our Solidity semantics.

7.1 Constant Folding (Constant_Folding)

theory *Constant_Folding*
imports
  Solidity_Main
begin

The following function optimizes expressions w.r.t. gas consumption.

```
primrec eupdate :: "E ⇒ E"
and lupdate :: "L ⇒ L"
where
  "lupdate (Id i) = Id i"
| "lupdate (Ref i exp) = Ref i (map eupdate exp)"
| "eupdate (E.INT b v) =
    (if (b∈vbits)
      then if v ≥ 0
        then E.INT b (-(2^(b-1)) + (v+2^(b-1)) mod (2^b))
        else E.INT b (2^(b-1) - (-v+2^(b-1)-1) mod (2^b) - 1)
      else E.INT b v)"
| "eupdate (UINT b v) = (if (b∈vbits) then UINT b (v mod (2^b)) else UINT b v)"
| "eupdate (ADDRESS a) = ADDRESS a"
| "eupdate (BALANCE a) = BALANCE a"
| "eupdate THIS = THIS"
| "eupdate SENDER = SENDER"
| "eupdate VALUE = VALUE"
| "eupdate TRUE = TRUE"
| "eupdate FALSE = FALSE"
| "eupdate (LVAL l) = LVAL (lupdate l)"
| "eupdate (PLUS ex1 ex2) =
    (case (eupdate ex1) of
      E.INT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            E.INT b2 v2 ⇒
              if b2∈vbits
                then let v=v1+v2 in
                  if v ≥ 0
                    then E.INT (max b1 b2) (-(2^((max b1 b2)-1)) + (v+2^((max b1 b2)-1)) mod (2^(max b1
b2)))
                    else E.INT (max b1 b2) (2^((max b1 b2)-1) - (-v+2^((max b1 b2)-1)-1) mod (2^(max b1
b2)) - 1)
                else (PLUS (E.INT b1 v1) (E.INT b2 v2))
          | UINT b2 v2 ⇒
              if b2∈vbits ∧ b2 < b1
                then let v=v1+v2 in
                  if v ≥ 0
                    then E.INT b1 (-(2^(b1-1)) + (v+2^(b1-1)) mod (2^b1))
                    else E.INT b1 (2^(b1-1) - (-v+2^(b1-1)-1) mod (2^b1) - 1)
                else PLUS (E.INT b1 v1) (UINT b2 v2)
          | _ ⇒ PLUS (E.INT b1 v1) (eupdate ex2))
        else PLUS (E.INT b1 v1) (eupdate ex2)
    | UINT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
```

```
            UINT b2 v2 ⇒
              if b2 ∈ vbits
                then UINT (max b1 b2) ((v1 + v2) mod (2^(max b1 b2)))
                else (PLUS (UINT b1 v1) (UINT b2 v2))
          | E.INT b2 v2 ⇒
              if b2∈vbits ∧ b1 < b2
                then let v=v1+v2 in
                  if v ≥ 0
                    then E.INT b2 (-(2^(b2-1)) + (v+2^(b2-1)) mod (2^b2))
                    else E.INT b2 (2^(b2-1) - (-v+2^(b2-1)-1) mod (2^b2) - 1)
                else PLUS (UINT b1 v1) (E.INT b2 v2)
          | _ ⇒ PLUS (UINT b1 v1) (eupdate ex2))
        else PLUS (UINT b1 v1) (eupdate ex2)
    | _ ⇒ PLUS (eupdate ex1) (eupdate ex2))"
| "eupdate (MINUS ex1 ex2) =
    (case (eupdate ex1) of
      E.INT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            E.INT b2 v2 ⇒
              if b2∈vbits
                then let v=v1-v2 in
                  if v ≥ 0
                    then E.INT (max b1 b2) (-(2^((max b1 b2)-1)) + (v+2^((max b1 b2)-1)) mod (2^(max b1
b2)))
                    else E.INT (max b1 b2) (2^((max b1 b2)-1) - (-v+2^((max b1 b2)-1)-1) mod (2^(max b1
b2)) - 1)
                else (MINUS (E.INT b1 v1) (E.INT b2 v2))
          | UINT b2 v2 ⇒
              if b2∈vbits ∧ b2 < b1
                then let v=v1-v2 in
                  if v ≥ 0
                    then E.INT b1 (-(2^(b1-1)) + (v+2^(b1-1)) mod (2^b1))
                    else E.INT b1 (2^(b1-1) - (-v+2^(b1-1)-1) mod (2^b1) - 1)
                else MINUS (E.INT b1 v1) (UINT b2 v2)
          | _ ⇒ MINUS (E.INT b1 v1) (eupdate ex2))
        else MINUS (E.INT b1 v1) (eupdate ex2)
    | UINT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            UINT b2 v2 ⇒
              if b2 ∈ vbits
                then UINT (max b1 b2) ((v1 - v2) mod (2^(max b1 b2)))
                else (MINUS (UINT b1 v1) (UINT b2 v2))
          | E.INT b2 v2 ⇒
              if b2∈vbits ∧ b1 < b2
                then let v=v1-v2 in
                  if v ≥ 0
                    then E.INT b2 (-(2^(b2-1)) + (v+2^(b2-1)) mod (2^b2))
                    else E.INT b2 (2^(b2-1) - (-v+2^(b2-1)-1) mod (2^b2) - 1)
                else MINUS (UINT b1 v1) (E.INT b2 v2)
          | _ ⇒ MINUS (UINT b1 v1) (eupdate ex2))
        else MINUS (UINT b1 v1) (eupdate ex2)
    | _ ⇒ MINUS (eupdate ex1) (eupdate ex2))"
| "eupdate (EQUAL ex1 ex2) =
    (case (eupdate ex1) of
      E.INT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            E.INT b2 v2 ⇒
              if b2∈vbits
                then if v1 = v2
                  then TRUE
                  else FALSE
```

```
                else EQUAL (E.INT b1 v1) (E.INT b2 v2)
          | UINT b2 v2 ⇒
              if b2∈vbits ∧ b2 < b1
                then if v1 = v2
                  then TRUE
                  else FALSE
                else EQUAL (E.INT b1 v1) (UINT b2 v2)
          | _ ⇒ EQUAL (E.INT b1 v1) (eupdate ex2))
        else EQUAL (E.INT b1 v1) (eupdate ex2)
    | UINT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            UINT b2 v2 ⇒
              if b2 ∈ vbits
                then if v1 = v2
                  then TRUE
                  else FALSE
                else EQUAL (E.INT b1 v1) (UINT b2 v2)
          | E.INT b2 v2 ⇒
              if b2∈vbits ∧ b1 < b2
                then if v1 = v2
                    then TRUE
                    else FALSE
                else EQUAL (UINT b1 v1) (E.INT b2 v2)
          | _ ⇒ EQUAL (UINT b1 v1) (eupdate ex2))
        else EQUAL (UINT b1 v1) (eupdate ex2)
    | _ ⇒ EQUAL (eupdate ex1) (eupdate ex2))"
| "eupdate (LESS ex1 ex2) =
    (case (eupdate ex1) of
      E.INT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            E.INT b2 v2 ⇒
              if b2∈vbits
                then if v1 < v2
                  then TRUE
                  else FALSE
                else LESS (E.INT b1 v1) (E.INT b2 v2)
          | UINT b2 v2 ⇒
              if b2∈vbits ∧ b2 < b1
                then if v1 < v2
                  then TRUE
                  else FALSE
                else LESS (E.INT b1 v1) (UINT b2 v2)
          | _ ⇒ LESS (E.INT b1 v1) (eupdate ex2))
        else LESS (E.INT b1 v1) (eupdate ex2)
    | UINT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            UINT b2 v2 ⇒
              if b2 ∈ vbits
                then if v1 < v2
                  then TRUE
                  else FALSE
                else LESS (E.INT b1 v1) (UINT b2 v2)
          | E.INT b2 v2 ⇒
              if b2∈vbits ∧ b1 < b2
                then if v1 < v2
                    then TRUE
                    else FALSE
                else LESS (UINT b1 v1) (E.INT b2 v2)
          | _ ⇒ LESS (UINT b1 v1) (eupdate ex2))
        else LESS (UINT b1 v1) (eupdate ex2)
    | _ ⇒ LESS (eupdate ex1) (eupdate ex2))"
```

```
| "eupdate (AND ex1 ex2) =
    (case (eupdate ex1) of
      TRUE ⇒ (case (eupdate ex2) of
                TRUE ⇒ TRUE
              | FALSE ⇒ FALSE
              | _ ⇒ AND TRUE (eupdate ex2))
    | FALSE ⇒ (case (eupdate ex2) of
                TRUE ⇒ FALSE
              | FALSE ⇒ FALSE
              | _ ⇒ AND FALSE (eupdate ex2))
    | _ ⇒ AND (eupdate ex1) (eupdate ex2))"
| "eupdate (OR ex1 ex2) =
    (case (eupdate ex1) of
      TRUE ⇒ (case (eupdate ex2) of
                TRUE ⇒ TRUE
              | FALSE ⇒ TRUE
              | _ ⇒ OR TRUE (eupdate ex2))
    | FALSE ⇒ (case (eupdate ex2) of
                TRUE ⇒ TRUE
              | FALSE ⇒ FALSE
              | _ ⇒ OR FALSE (eupdate ex2))
    | _ ⇒ OR (eupdate ex1) (eupdate ex2))"
| "eupdate (NOT ex1) =
    (case (eupdate ex1) of
      TRUE ⇒ FALSE
    | FALSE ⇒ TRUE
    | _ ⇒ NOT (eupdate ex1))"
| "eupdate (CALL i xs) = CALL i xs"
| "eupdate (ECALL e i xs r) = ECALL e i xs r"

value "eupdate (UINT 8 250)"
lemma "eupdate (UINT 8 250)
      =UINT 8 250"
  ⟨proof⟩
lemma "eupdate (UINT 8 500)
      = UINT 8 244"
  ⟨proof⟩
lemma "eupdate (E.INT 8 (-100))
      = E.INT 8 (- 100)"
  ⟨proof⟩
lemma "eupdate (E.INT 8 (-150))
      = E.INT 8 106"
  ⟨proof⟩
lemma "eupdate (PLUS (UINT 8 100) (UINT 8 100))
      = UINT 8 200"
  ⟨proof⟩
lemma "eupdate (PLUS (UINT 8 257) (UINT 16 100))
      = UINT 16 101"
  ⟨proof⟩
lemma "eupdate (PLUS (E.INT 8 100) (UINT 8 250))
      = PLUS (E.INT 8 100) (UINT 8 250)"
  ⟨proof⟩
lemma  "eupdate (PLUS (E.INT 8 250) (UINT 8 500))
      = PLUS (E.INT 8 (- 6)) (UINT 8 244)"
  ⟨proof⟩
lemma "eupdate (PLUS (E.INT 16 250) (UINT 8 500))
      = E.INT 16 494"
  ⟨proof⟩
lemma "eupdate (EQUAL (UINT 16 250) (UINT 8 250))
      = TRUE"
  ⟨proof⟩
lemma  "eupdate (EQUAL (E.INT 16 100) (UINT 8 100))
      = TRUE"
  ⟨proof⟩
```

lemma *"eupdate (EQUAL (E.INT 8 100) (UINT 8 100))*
= EQUAL (E.INT 8 100) (UINT 8 100)"
⟨*proof*⟩

lemma *update_bounds_int:*
assumes *"eupdate ex = (E.INT b v)"* **and** *"b∈vbits"*
shows *"(v < 2^(b-1)) ∧ v ≥ -(2^(b-1))"*
⟨*proof*⟩

lemma *update_bounds_uint:*
assumes *"eupdate ex = UINT b v"* **and** *"b∈vbits"*
shows *"v < 2^b ∧ v ≥ 0"*
⟨*proof*⟩

lemma *no_gas:*
assumes *"¬ gas st > 0"*
shows *"expr ex ep env cd st = Exception Gas"*
⟨*proof*⟩

lemma *lift_eq:*
assumes *"expr e1 ep env cd st = expr e1' ep env cd st"*
and *"⋀st' rv. expr e1 ep env cd st = Normal (rv, st') ⟹ expr e2 ep env cd st'= expr e2' ep env cd st'"*
shows *"lift expr f e1 e2 ep env cd st=lift expr f e1' e2' ep env cd st"*
⟨*proof*⟩

lemma *ssel_eq_ssel:*
"(⋀i st. i ∈ set ix ⟹ expr i ep env cd st = expr (f i) ep env cd st)
⟹ ssel tp loc ix ep env cd st = ssel tp loc (map f ix) ep env cd st"
⟨*proof*⟩

lemma *msel_eq_msel:*
"(⋀i st. i ∈ set ix ⟹ expr i ep env cd st = expr (f i) ep env cd st) ⟹
msel c tp loc ix ep env cd st = msel c tp loc (map f ix) ep env cd st"
⟨*proof*⟩

lemma *ref_eq:*
assumes *"⋀e st. e ∈ set ex ⟹ expr e ep env cd st = expr (f e) ep env cd st"*
shows *"rexp (Ref i ex) ep env cd st=rexp (Ref i (map f ex)) ep env cd st"*
⟨*proof*⟩

The following theorem proves that the update function preserves the semantics of expressions.

theorem *update_correctness:*
"⋀st lb lv. expr ex ep env cd st = expr (eupdate ex) ep env cd st"
"⋀st. rexp lv ep env cd st = rexp (lupdate lv) ep env cd st"
⟨*proof*⟩

end

7.2 Reentrancy (Reentrancy)

In the following we use our semantics to verify a contract implementing a simple token. The contract is defined by definition *victim* and consist of one state variable and two methods:

- The state variable "balance" is a mapping which assigns a balance to each address.
- Method "deposit" allows to send money to the contract which is then added to the sender's balance.
- Method "withdraw" allows to withdraw the callers balance.

We then verify that the following invariant (defined by *INV*) is preserved by both methods: The difference between

- the contracts own account-balance and
- the sum of all the balances kept in the contracts state variable is larger than a certain threshold.

There are two things to note here: First, 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 implict call to the callee's fallback method. This functionality was exploited in the infamous DAO attack which we demonstrate it in terms of an example later on. Since we do not know all potential contracts which call "withdraw", we need to verify our invariant for all possible Solidity programs. Thus, the core result here is a lemma which shows that the invariant is preserved by every Solidity program which is not executed in the context of our own contract. For our own methods we show that the invariant holds after executing it. Since our own program as well as the unknown program may depend on each other both properties are encoded in a single lemma (*secure*) which is then proved by induction over all statements. The final result is then given in terms of two corollaries for the corresponding methods of our contract.

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 just send us additional money withouth 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.

theory *Reentrancy*
imports *Solidity_Evaluator*
begin

7.2.1 Example of Re-entrancy

```
value "eval 1000
          stmt
          (COMP
            (EXTERNAL (ADDRESS (STR ''Victim'')) (STR ''deposit'') [] (UINT 256 10))
            (EXTERNAL (ADDRESS (STR ''Victim'')) (STR ''withdraw'') [] (UINT 256 0)))
          (STR ''Attacker'')
          (STR '''')
          (STR ''0'')
          [(STR ''Victim'', STR ''100''), (STR ''Attacker'', STR ''100'')]
          [
            (STR ''Attacker'',
              [],
              ITE
                (LESS (BALANCE THIS) (UINT 256 125))
                (EXTERNAL (ADDRESS (STR ''Victim'')) (STR ''withdraw'') [] (UINT 256 0))
                SKIP),
            (STR ''Victim'',
              [
                (STR ''balance'', Var (STMap TAddr (STValue (TUInt 256)))),
                (STR ''deposit'', Method ([], ASSIGN (Ref (STR ''balance'') [SENDER]) VALUE, None)),
                (STR ''withdraw'', Method ([],
                ITE
                  (LESS (UINT 256 0) (LVAL (Ref (STR ''balance'') [SENDER])))
                  (COMP
                    (TRANSFER SENDER (LVAL (Ref (STR ''balance'') [SENDER])))
                    (ASSIGN (Ref (STR ''balance'') [SENDER]) (UINT 256 0)))
                  SKIP
                , None))],
              SKIP)
          ]
          []"
```

7.2.2 Definition of Contract

abbreviation *myrexp::L*
  where *"myrexp ≡ Ref (STR ''balance'') [SENDER]"*

abbreviation *mylval::E*
  where *"mylval ≡ LVAL myrexp"*

abbreviation *assign::S*
  where *"assign ≡ ASSIGN (Ref (STR ''balance'') [SENDER]) (UINT 256 0)"*

abbreviation *transfer::S*
  where *"transfer ≡ TRANSFER SENDER (LVAL (Id (STR ''bal'')))"*

abbreviation *comp::S*
  where *"comp ≡ COMP assign transfer"*

abbreviation *keep::S*
  where *"keep ≡ BLOCK ((STR ''bal'', Value (TUInt 256)), Some mylval) comp"*

abbreviation *deposit::S*
  where *"deposit ≡ ASSIGN (Ref (STR ''balance'') [SENDER]) (PLUS (LVAL (Ref (STR ''balance'') [SENDER])) VALUE)"*

definition *victim::"(Identifier, Member) fmap"*
  where *"victim ≡ fmap_of_list [*
(STR ''balance'', Var (STMap TAddr (STValue (TUInt 256)))),
(STR ''deposit'', Method ([], deposit, None)),
(STR ''withdraw'', Method ([], keep, None))]"

7.2.3 Definition of Invariant

abbreviation *"SUMM s ≡ ∑(ad,x)|fmlookup s (ad + (STR ''.'' + STR ''balance'')) = Some x. ReadL$_{int}$ x"*

abbreviation *"POS s ≡ ∀ad x. fmlookup s (ad + (STR ''.'' + STR ''balance'')) = Some x ⟶ ReadL$_{int}$ x ≥ 0"*

abbreviation *"INV st s val bal ≡*
  fmlookup (storage st) (STR ''Victim'') = Some s ∧ ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - val ≥ bal ∧ bal ≥ 0"

definition *frame_def: "frame bal st ≡ (∃s. INV st s (SUMM s) bal ∧ POS s)"*

7.2.4 Verification

lemma *conj3: "P ⟹ Q ⟹ R ⟹ P ∧ (Q ∧ R)"* ⟨*proof*⟩

lemma *fmfinite: "finite ({(ad, x). fmlookup y ad = Some x})"*
⟨*proof*⟩

lemma *fmlookup_finite:*
  fixes *f :: "'a ⇒ 'a"*
    and *y :: "('a, 'b) fmap"*
  assumes *"inj_on (λ(ad, x). (f ad, x)) {(ad, x). (fmlookup y ∘ f) ad = Some x}"*
  shows *"finite {(ad, x). (fmlookup y ∘ f) ad = Some x}"*
⟨*proof*⟩

lemma *balance_inj: "inj_on (λ(ad, x). (ad + (STR ''.'' + STR ''balance''),x)) {(ad, x). (fmlookup y ∘ f) ad = Some x}"*
⟨*proof*⟩

lemma *transfer_frame:*
assumes *"Accounts.transfer ad adv v (accounts st) = Some acc"*
    and *"frame bal st"*
    and *"ad ≠ STR ''Victim''"*
  shows *"frame bal (st(|accounts := acc|))"*
⟨*proof*⟩

lemma *decl_frame:*

assumes *"frame bal st"*
and *"decl a1 a2 a3 cp cd mem c env st = Normal (rv, st')"*
shows *"frame bal st'"*
⟨*proof*⟩

context *statement_with_gas*
begin

lemma *secureassign:*
assumes *"stmt assign ep env cd st = Normal((), st')"*
and *"fmlookup (storage st) (STR ''Victim'') = Some s"*
and *"address env = (STR ''Victim'')"*
and *"fmlookup (denvalue env) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc STR ''balance'')"*
and *"accessStore x (stack st) = Some (KValue (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s))"*
and *"ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - (SUMM s) ≥ bal"*
and *"POS s"*
obtains *s'*
where *"fmlookup (storage st') (STR ''Victim'') = Some s'"*
and *"ReadL$_{int}$ (accessBalance (accounts st') (STR ''Victim'')) - (SUMM s' + ReadL$_{int}$ (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s)) ≥ bal"*
and *"accessStore x (stack st') = Some (KValue (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s))"*
and *"POS s'"*
⟨*proof*⟩

lemma *securesender:*
assumes *"expr SENDER ep env cd st = Normal((KValue v,t), st')"*
and *"fmlookup (storage st) (STR ''Victim'') = Some s"*
and *"ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - SUMM s ≥ bal ∧ POS s"*
obtains *s'* **where**
"v = sender env"
and *"t = Value TAddr"*
and *"fmlookup (storage st') (STR ''Victim'') = Some s'"*
and *"ReadL$_{int}$ (accessBalance (accounts st') (STR ''Victim'')) - SUMM s' ≥ bal ∧ POS s'"*
⟨*proof*⟩

lemma *securessel:*
assumes *"ssel type loc [] ep env cd st = Normal (x, st')"*
and *"fmlookup (storage st) (STR ''Victim'') = Some s"*
and *"ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - SUMM s ≥ bal ∧ POS s"*
obtains *s'* **where**
"x = (loc, type)"
and *"fmlookup (storage st') (STR ''Victim'') = Some s'"*
and *"ReadL$_{int}$ (accessBalance (accounts st') (STR ''Victim'')) - SUMM s' ≥ bal ∧ POS s'"*
⟨*proof*⟩

lemma *securessel2:*
assumes *"ssel (STMap TAddr (STValue (TUInt 256))) (STR ''balance'') [SENDER] ep env cd st = Normal ((loc, type), st')"*
and *"fmlookup (storage st) (STR ''Victim'') = Some s"*
and *"ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - SUMM s ≥ bal ∧ POS s"*
obtains *s'* **where**
"loc = sender env + (STR ''.'' + STR ''balance'')"
and *"type = STValue (TUInt 256)"*
and *"fmlookup (storage st') (STR ''Victim'') = Some s'"*
and *"ReadL$_{int}$ (accessBalance (accounts st') (STR ''Victim'')) - SUMM s' ≥ bal ∧ POS s'"*
⟨*proof*⟩

lemma *securerexp:*
assumes *"rexp myrexp e$_p$ env cd st = Normal ((v, t), st')"*

and *"fmlookup (denvalue env) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc STR ''balance'')"*
and *"fmlookup (storage st) (STR ''Victim'') = Some s"*
and *"ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - SUMM s ≥ bal ∧ POS s"*
and *"address env = STR ''Victim''"*
obtains *s'* **where**
"fmlookup (storage st') (address env) = Some s'"
and *"v = KValue (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s')"*
and *"t = Value (TUInt 256)"*
and *"ReadL$_{int}$ (accessBalance (accounts st') (STR ''Victim'')) - SUMM s' ≥ bal ∧ POS s'"*
⟨*proof*⟩

lemma *securelval:*
assumes *"expr mylval ep env cd st = Normal((v,t), st')"*
and *"fmlookup (denvalue env) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc STR ''balance'')"*
and *"fmlookup (storage st) (STR ''Victim'') = Some s"*
and *"ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - SUMM s ≥ bal ∧ bal ≥ 0 ∧ POS s"*
and *"address env = STR ''Victim''"*
obtains *s'* **where** *"fmlookup (storage st') (STR ''Victim'') = Some s'"*
and *"v = KValue (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s')"*
and *"t = Value (TUInt 256)"*
and *"ReadL$_{int}$ (accessBalance (accounts st') (STR ''Victim'')) - SUMM s' ≥ bal ∧ bal ≥ 0 ∧ POS s'"*
⟨*proof*⟩

lemma *plus_frame:*
assumes *"expr (PLUS (LVAL (Ref (STR ''balance'') [SENDER])) VALUE) ep env cd st = Normal (kv, st')"*
and *"ReadL$_{int}$ (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s) + ReadL$_{int}$ (svalue env) < 2^256"*
and *"ReadL$_{int}$ (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s) + ReadL$_{int}$ (svalue env) ≥ 0"*
and *"fmlookup (storage st) (STR ''Victim'') = Some s"*
and *"ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - SUMM s ≥ bal"*
and *"fmlookup (denvalue env) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc STR ''balance'')"*
and *"address env = (STR ''Victim'')"*
shows *"kv = (KValue (ShowL$_{int}$ (ReadL$_{int}$ (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s) + ReadL$_{int}$ (svalue env))), Value (TUInt 256))"*
and *"fmlookup (storage st') (STR ''Victim'') = Some s"*
and *"ReadL$_{int}$ (accessBalance (accounts st') (STR ''Victim'')) = ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim''))"*
⟨*proof*⟩

lemma *deposit_frame:*
assumes *"stmt deposit ep env cd st = Normal ((), st')"*
and *"fmlookup (storage st) (STR ''Victim'') = Some s"*
and *"address env = (STR ''Victim'')"*
and *"fmlookup (denvalue env) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc STR ''balance'')"*
and *"ReadL$_{int}$ (accessBalance (accounts st) (STR ''Victim'')) - SUMM s ≥ bal + ReadL$_{int}$ (svalue env)"*
and *"ReadL$_{int}$ (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s) + ReadL$_{int}$ (svalue env) < 2^256"*
and *"ReadL$_{int}$ (accessStorage (TUInt 256) (sender env + (STR ''.'' + STR ''balance'')) s) + ReadL$_{int}$ (svalue env) ≥ 0"*
and *"POS s"*
obtains *s'*
where *"fmlookup (storage st') (STR ''Victim'') = Some s'"*
and *"ReadL$_{int}$ (accessBalance (accounts st') (STR ''Victim'')) - SUMM s' ≥ bal"*
and *"POS s'"*
⟨*proof*⟩

lemma *secure:*
"address ev1 ≠ (STR ''Victim'') ∧ fmlookup ep1 (STR ''Victim'') = Some (victim, SKIP) ⟶ (∀rv1 st1' bal. frame bal st1 ∧ msel c1 t1 l1 xe1 ep1 ev1 cd1 st1 = Normal (rv1, st1') ⟶ frame bal st1')"
"address ev2 ≠ (STR ''Victim'') ∧ fmlookup ep2 (STR ''Victim'') = Some (victim, SKIP) ⟶ (∀rv2 st2' bal. frame bal st2 ∧ ssel t2 l2 xe2 ep2 ev2 cd2 st2 = Normal (rv2, st2') ⟶ frame bal st2')"
"address ev5 ≠ (STR ''Victim'') ∧ fmlookup ep5 (STR ''Victim'') = Some (victim, SKIP) ⟶ (∀rv3 st5' bal. frame bal st5 ∧ lexp l5 ep5 ev5 cd5 st5 = Normal (rv3, st5') ⟶ frame bal st5')"
"address ev4 ≠ (STR ''Victim'') ∧ fmlookup ep4 (STR ''Victim'') = Some (victim, SKIP) ⟶ (∀rv4 st4' bal. frame bal st4 ∧ expr e4 ep4 ev4 cd4 st4 = Normal (rv4, st4') ⟶ frame bal st4')"
"address lev ≠ (STR ''Victim'') ∧ fmlookup lep (STR ''Victim'') = Some (victim, SKIP) ⟶ (∀ev cd st st' bal. load lcp lis lxs lep lev0 lcd0 lst0 lev lcd lst = Normal ((ev, cd, st), st') ⟶ (frame bal lst0 ⟶ frame bal st) ∧ (frame bal lst ⟶ frame bal st') ∧ address lev0 = address ev ∧ sender lev0 = sender ev ∧ svalue lev0 = svalue ev)"
"address ev3 ≠ (STR ''Victim'') ∧ fmlookup ep3 (STR ''Victim'') = Some (victim, SKIP) ⟶ (∀rv3 st3' bal. frame bal st3 ∧ rexp l3 ep3 ev3 cd3 st3 = Normal (rv3, st3') ⟶ frame bal st3')"
"(fmlookup ep6 (STR ''Victim'') = Some (victim, SKIP) ⟶
(∀st6'. stmt s6 ep6 ev6 cd6 st6 = Normal((), st6') ⟶
((address ev6 ≠ (STR ''Victim'') ⟶ (∀bal. frame bal st6 ⟶ frame bal st6'))
∧ (address ev6 = (STR ''Victim'') ⟶
(∀s val bal x. s6 = transfer
∧ INV st6 s (SUMM s + ReadL$_{int}$ val) bal ∧ POS s
∧ fmlookup (denvalue ev6) (STR ''bal'') = Some (Value (TUInt 256), Stackloc x)
∧ accessStore x (stack st6) = Some (KValue val)
∧ sender ev6 ≠ address ev6
⟶ (∃s'. fmlookup (storage st6') (STR ''Victim'') = Some s'
∧ ReadL$_{int}$ (accessBalance (accounts st6') (STR ''Victim'')) - (SUMM s') ≥ bal ∧ bal ≥ 0 ∧ POS s')) ∧
(∀s bal x. s6 = comp
∧ INV st6 s (SUMM s) bal ∧ POS s
∧ fmlookup (denvalue ev6) (STR ''bal'') = Some (Value (TUInt 256), Stackloc x)
∧ fmlookup (denvalue ev6) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc STR ''balance'')
∧ accessStore x (stack st6) = Some (KValue (accessStorage (TUInt 256) (sender ev6 + (STR ''.'' + STR ''balance'')) s))
∧ sender ev6 ≠ address ev6
⟶ (∃s'. fmlookup (storage st6') (STR ''Victim'') = Some s'
∧ ReadL$_{int}$ (accessBalance (accounts st6') (STR ''Victim'')) - (SUMM s') ≥ bal ∧ bal ≥ 0 ∧ POS s')) ∧
(∀s bal. s6 = keep
∧ INV st6 s (SUMM s) bal ∧ POS s
∧ fmlookup (denvalue ev6) (STR ''balance'') = Some (Storage (STMap TAddr (STValue (TUInt 256))), Storeloc STR ''balance'')
∧ sender ev6 ≠ address ev6
⟶ (∃s'. fmlookup (storage st6') (STR ''Victim'') = Some s'
∧ ReadL$_{int}$ (accessBalance (accounts st6') (STR ''Victim'')) - (SUMM s') ≥ bal ∧ bal ≥ 0 ∧ POS s'))
))))"
⟨*proof*⟩

corollary *final1:*
assumes "fmlookup ep (STR ''Victim'') = Some (victim, SKIP)"
and "stmt (EXTERNAL (ADDRESS (STR ''Victim'')) (STR ''withdraw'') [] val) ep env cd st = Normal((), st')"
and "address env ≠ (STR ''Victim'')"
and "frame bal st"
shows "frame bal st'"
⟨*proof*⟩

corollary *final2:*
assumes "fmlookup ep (STR ''Victim'') = Some (victim, SKIP)"

and *"stmt (EXTERNAL (ADDRESS (STR ''Victim'')) (STR ''deposit'') [] val) ep env cd st = Normal((), st')"*
and *"address env ≠ (STR ''Victim'')"*
and *"frame bal st"*
shows *"frame bal st'"*
⟨*proof*⟩

end

end

Bibliography

[1] The Bitcon market capitalisation. URL https://coinmarketcap.com/currencies/bitcoin/. Last checked on 2021-05-04.

[2] D. Marmsoler and A. D. Brucker. A denotational semantics of Solidity in Isabelle/HOL. In R. Calinescu and C. Pasareanu, editors, *Software Engineering and Formal Methods (SEFM)*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2021. ISBN 3-540-25109-X. URL https://www.brucker.ch/bibliography/abstract/marmsoler.ea-solidity-semantics-2021.

[3] D. Marmsoler and A. D. Brucker. Conformance testing of formal semantics using grammar-based fuzzing. In L. Kovacs and K. Meinke, editors, *TAP 2022: Tests And Proofs*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2022. ISBN 978-3-642-38915-3. URL https://www.brucker.ch/bibliography/abstract/marmsoler.ea-conformance-2022.

[4] Online. Solidity documentation. https://solidity.readthedocs.io/en/latest.

[5] D. Perez and B. Livshits. Smart contract vulnerabilities: Vulnerable does not imply exploited. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/perez.

[6] G. Wood et al. Ethereum: A secure decentralised generalised transaction ledger, 2022. Berlin Version 3078285 – 2022-07-13. https://ethereum.github.io/yellowpaper/paper.pdf.