

Isabelle/Solidity

A deep Embedding of Solidity in Isabelle/HOL

Diego Marmsoler^{id} and Achim D. Brucker^{id} and Billy Thornton

June 15, 2026

Department of Computer Science, University of Exeter, Exeter, UK
{d.marmsoler, a.brucker, bt319}@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	Some Basic Lemmas for Finite Maps (Utils)	15
2.3	Some Basic Proof Methods (Utils)	15
2.4	state Monad with Exceptions (StateMonad)	15
2.5	Hoare Logic (StateMonad)	18
3	Types and Accounts	21
3.1	Value types (Valuetypes)	21
3.2	Accounts (Accounts)	27
4	Stores and Environment	31
4.1	Storage (Storage)	31
4.2	Environment and state (Environment)	37
5	Expressions and Statements	45
5.1	Contracts (Contracts)	45
5.2	Expressions (Expressions)	50
5.3	Statements (Statements)	57
5.4	Examples (Statements)	69
5.5	The Main Entry Point (Solidity_Main)	70
6	A Solidity Evaluation System	71
6.1	Towards a Setup for Symbolic Evaluation of Solidity (Solidity_Symbex)	71
6.2	Solidity Evaluator and Code Generator Setup (Solidity_Evaluator)	71
6.3	Generating an Executable of the Evaluator (Compile_Evaluator)	83
7	Verification Support	85
7.1	Setup for Monad VCG (Weakest_Precondition)	85
7.2	Calculus (Weakest_Precondition)	86
7.3	Verification Condition Generator (Weakest_Precondition)	98
8	Applications	103
8.1	Reentrancy (Reentrancy)	103
8.2	Constant Folding (Constant_Folding)	110

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).

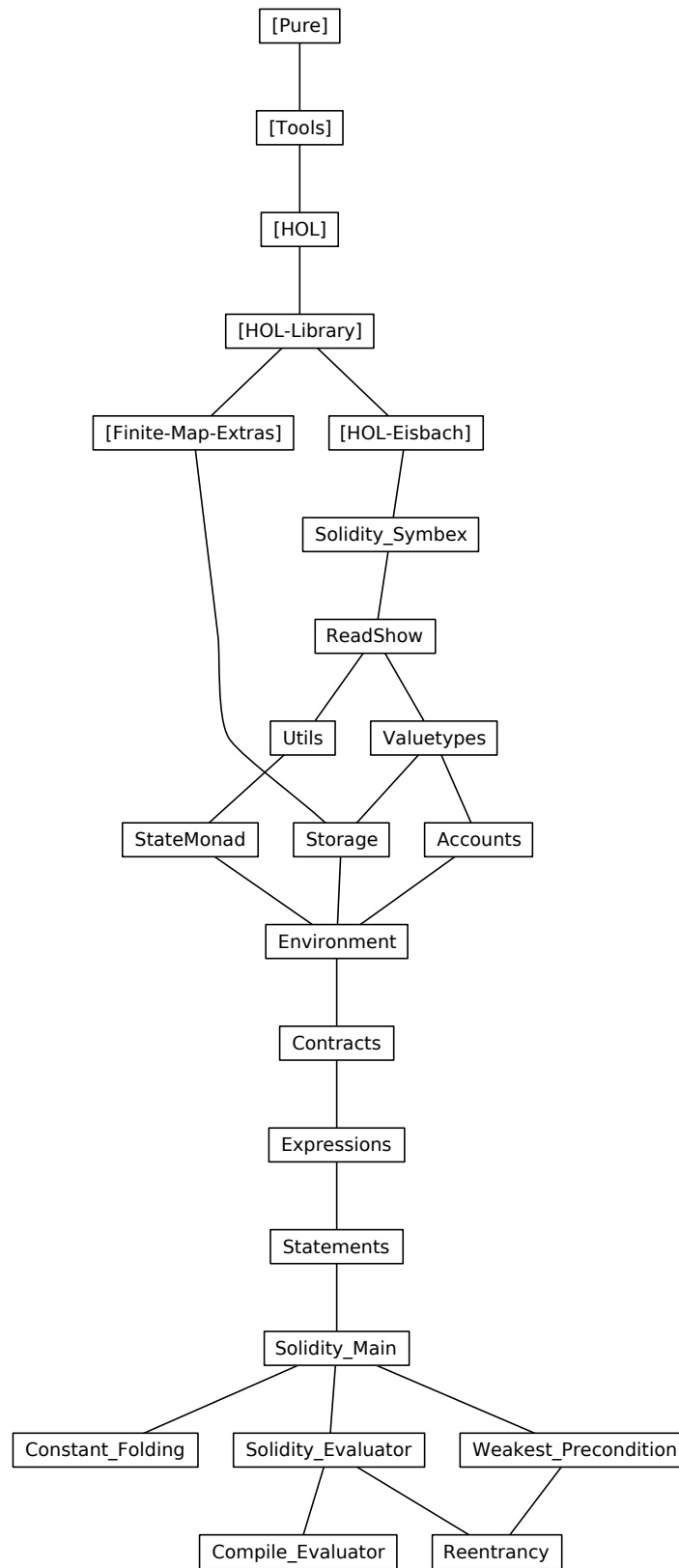


Figure 1.1: The Dependency Graph of the Isabelle Theories.

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

```
<Readbool s = (if s = ''True'' then True else False)>
```

definition

```
<Showbool b = (if b then ''True'' else ''False'')>
```

definition

```
<STR_is_bool s = (Showbool (Readbool s) = s)>
```

```
declare Readbool_def [solidity_symbex]
      Showbool_def [solidity_symbex]
```

```
lemma Show_Read_bool_id: <STR_is_bool s  $\implies$  (Showbool (Readbool s) = s)>
  <proof>
```

```
lemma STR_is_bool_split: <STR_is_bool s  $\implies$  s = ''False''  $\vee$  s = ''True''>
  <proof>
```

```
lemma Read_Show_bool_id: <Readbool (Showbool b) = b>
  <proof>
```

```
definition ReadLbool::<String.literal  $\implies$  bool> (<[_]>) where
```

```
<ReadLbool s = (if s = STR ''True'' then True else False)>
```

```
definition ShowLbool::<bool  $\implies$  String.literal> (<[_]>) where
```

```
<ShowLbool b = (if b then STR ''True'' else STR ''False'')>
```

definition

```
<strL_is_bool' s = (ShowLbool (ReadLbool s) = s)>
```

```
declare ReadLbool_def [solidity_symbex]
      ShowLbool_def [solidity_symbex]
```

```
lemma Show_Read_bool'_id: <strL_is_bool' s  $\implies$  (ShowLbool (ReadLbool s) = s)>
  <proof>
```

```
lemma strL_is_bool'_split: <strL_is_bool' s  $\implies$  s = STR ''False''  $\vee$  s = STR ''True''>
  <proof>
```

```
lemma Read_Show_bool'_id[simp]: <ReadLbool (ShowLbool b) = b>
  <proof>
```

```
lemma true_neq_false[simp]: "ShowLbool True  $\neq$  ShowLbool False"
  <proof>
```

Natural Numbers

```
definition nat_of_digit :: <char  $\Rightarrow$  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  $\Rightarrow$  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)>
```

```
definition is_digit :: <char  $\Rightarrow$  bool> where
  <is_digit c =
    (if c = CHR ''0'' then True
     else if c = CHR ''1'' then True
     else if c = CHR ''2'' then True
     else if c = CHR ''3'' then True
     else if c = CHR ''4'' then True
     else if c = CHR ''5'' then True
     else if c = CHR ''6'' then True
     else if c = CHR ''7'' then True
     else if c = CHR ''8'' then True
     else if c = CHR ''9'' then True
     else False)>
```

```
declare digit_of_nat_def [solidity_symbex]
```

```
lemma nat_of_digit_digit_of_nat_id:
  <x < 10  $\implies$  nat_of_digit (digit_of_nat x) = x>
  <proof>
```

```
lemma img_digit_of_nat:
  <n < 10  $\implies$  digit_of_nat n  $\in$  {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) (indexed_from 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) (indexed_from (Suc 0) (rev a))) 0
  >
  <proof>

lemma indexed_from_Suc_eq': <indexed_from (Suc n) 1 = map (λ (a,b). (a+1::nat,b)) (indexed_from n 1)>

```

2 Preliminaries

<proof>

lemma *mult_assoc_aux1*:

*<($\lambda(p, y). 10^p * y$) \circ ($\lambda(a, y). (\text{Suc } a, y)$) = ($\lambda(p, y). (10::\text{nat}) * (10^p * y)$)>*
<proof>

lemma *fold_map_transfer*:

*<(foldr (+) (map ($\lambda(x,y). 10 * (f (x,y))$)) 1) (0::nat)) = 10 * (foldr (+) (map ($\lambda x. (f x)$) 1) (0::nat))>*
<proof>

lemma *mult_assoc_aux2*: *<($\lambda(p, y). 10 * 10^p * (y::\text{nat})$) = ($\lambda(p, y). 10 * (10^p * y)$)>*
<proof>

lemma *nat_implode_explode_id*: *<nat_implode (nat_explode n) = n>*
<proof>

definition

Read_{nat} :: <string \Rightarrow nat> where
<Read_{nat} s = nat_implode (map nat_of_digit s)>

definition

Show_{nat} :: "nat \Rightarrow string" where
<Show_{nat} n = map digit_of_nat (nat_explode n)>

definition *is_nat* :: "string \Rightarrow bool" **where**

"is_nat s = (if s = [] then False else foldl (\wedge) True (map is_digit s))"

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) \subseteq {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) \neq []>*

<proof>

lemma *not_hd*: *<L \neq [] \implies e \notin set(L) \implies hd L \neq e>*

<proof>

lemma *Show_nat_not_neg''*: *<hd (Show_{nat} n) \neq (CHR '-''>*

<proof>

lemma *Show_Read_nat_id*: *<STR_is_nat s \implies (Show_{nat} (Read_{nat} s) = s)>*

<proof>

lemma *bar'*: *< $\forall d \in \text{set } l. d < 10 \implies \text{map nat_of_digit (map digit_of_nat } l) = l$ >*

<proof>

lemma *Read_Show_nat_id*: *<Read_{nat} (Show_{nat} n) = n>*

<proof>

definition

```
ReadLnat :: <String.literal ⇒ nat> (<[_]>) where
<ReadLnat = Readnat ∘ String.explode>
```

definition

```
ShowLnat :: <nat ⇒ String.literal> (<[_]>) where
<ShowLnat = String.implode ∘ Shownat>
```

```
declare ReadLnat_def [solidity_symbex]
      ShowLnat_def [solidity_symbex]
```

```
lemma ShowLNatDot: "CHR '.' ∉ set(String.explode (ShowLnat i))"
  <proof>
```

```
lemma ShowNatDot: "CHR '.' ∉ set(Shownat i) "
  <proof>
```

definition

```
<strL_is_nat' s = (ShowLnat (ReadLnat s) = s)>
```

```
value <[STR ''10'']::nat>
value <ReadLnat (STR ''10'')>
value <[10::nat]>
value <ShowLnat 10>
value <ReadLnat (ShowLnat (10)) = 10>
value <ShowLnat (ReadLnat (STR ''10'')) = STR ''10'')>
```

```
lemma Show_Read_nat'_id: <strL_is_nat' s ⇒ (ShowLnat (ReadLnat 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 Shownat_ascii: <map String.ascii_of (Shownat n) = Shownat n>
  <proof>
```

```
lemma Read_Show_nat'_id: <ReadLnat (ShowLnat n) = n>
  <proof>
```

Integer**definition**

```
Readint :: <string ⇒ int> where
<Readint x = (if hd x = (CHR ''-'') then -(int (Readnat (tl x))) else int (Readnat x))>
```

definition

```
Showint :: <int ⇒ string> where
<Showint i = (if i < 0 then (CHR ''-'')#(Shownat (nat (-i)))
  else Shownat (nat i))>
```

definition

```
<STR_is_int s = (Showint (Readint s) = s)>
```

```

declare Readint_def [solidity_symbex]
      Showint_def [solidity_symbex]

value <Readint (Showint 10) = 10>
value <Readint (Showint (-10)) = -10>

value <Showint (Readint (''10'')) = ''10''>
value <Showint (Readint (''-0'')) = ''-0''>
value "Showint (Readint (''00'')) "

definition
is_uint :: <string ⇒ bool> where
  <is_uint s = (if hd s = (CHR ''-'') then False else
    if hd s = (CHR ''0'') ∧ tl s ≠ [] then False else (is_nat s))>

value "is_uint ''07''"

definition
<STR_is_uint s = is_uint s>

lemma Show_Read_id: <STR_is_int s ⇒ (Showint (Readint s) = s)>
  <proof>

lemma Read_Show_id: <Readint (Showint x) = x>
  <proof>

lemma STR_is_int_Show: <STR_is_int (Showint n)>
  <proof>

definition
ReadLint :: <String.literal ⇒ int> (<[_]>) where
  <ReadLint = Readint ∘ String.explode>

definition
ShowLint :: <int ⇒ String.literal> (<[_]>) where
  <ShowLint = String.implode ∘ Showint>

definition
<strL_is_int' s = (ShowLint (ReadLint s) = s)>

declare ReadLint_def [solidity_symbex]
      ShowLint_def [solidity_symbex]

value <ReadLint (ShowLint 10) = 10>
value <ReadLint (ShowLint (-10)) = -10>

value <ShowLint (ReadLint (STR ''10'')) = STR ''10''>
value <ShowLint (ReadLint (STR ''-10'')) = STR ''-10''>

value <ShowLint (ReadLint (STR ''-000''))>

lemma Show_ReadL_id: <strL_is_int' s ⇒ (ShowLint (ReadLint s) = s)>
  <proof>

lemma Read_ShowL_id: <ReadLint (ShowLint x) = x>
  <proof>

lemma STR_is_int_ShowL: <strL_is_int' (ShowLint n)>
  <proof>

lemma String_Cancel: "a + (c::String.literal) = b + c ⇒ a = b"

```

<proof>

```
lemma ShowLintDot: "CHR '`.`' ∉ set(String.explode (ShowLint i))"
  <proof>
```

```
end
theory Utils
imports
  Main
  ReadShow
  "HOL-Library.Finite_Map"
  "HOL-Eisbach.Eisbach"
begin
```

2.2 Some Basic Lemmas for Finite Maps (Utils)

```
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 + z::String.literal,x)) {(ad, x). (fmlookup y ∘ f) ad = Some x}"
  <proof>
```

2.3 Some Basic Proof Methods (Utils)

```
method solve methods m = (m ; fail)

named_theorems intros
declare conjI[intros] impI[intros] allI[intros] ballI[intros]
method intros = (rule intros; intros?)

named_theorems elims
method elims = ((rule intros | erule elims); elims?)
declare conjE[elimis]

end
theory StateMonad
imports Main "HOL-Library.Monad_Syntax" Utils Solidity_Symbex
begin
```

2.4 state Monad with Exceptions (StateMonad)

```
datatype ('n, 'e) result = Normal (normal: 'n) | Exception (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>
```

2.4.1 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>
```

2.4.2 The Monad Laws

`return` is absorbed at the left of a ($\gg=$), applying the return value directly:

```
lemma return_bind [simp]: "(return x ≫= f) = f x"
<proof>
```

`return` is absorbed on the right of a ($\gg=$)

```
lemma bind_return [simp]: "(m ≫= return) = m"
<proof>
```

($\gg=$) 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>
```

2.4.3 Basic Congruence 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 " $\bigwedge a. x = \text{Some } a \implies f a s = f' a s$ " and " $x = \text{None} \implies g s = g' s$ "
  shows "(case x of Some a  $\implies$  f a | None  $\implies$  g) s = (case x' of Some a  $\implies$  f' a | None  $\implies$  g') s"
  <proof>

```

```

lemma bind_case_bool_cong [fundef_cong]:
  assumes "x = x'" and " $x = \text{True} \implies f s = f' s$ " and " $x = \text{False} \implies g s = g' s$ "
  shows "(case x of True  $\implies$  f | False  $\implies$  g) s = (case x' of True  $\implies$  f' | False  $\implies$  g') s"
  <proof>

```

2.4.4 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  $\implies$  (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  $\implies$  'a)  $\implies$  ('a, 'e, 's) state_monad" where
  "applyf f = get  $\gg$  ( $\lambda s. \text{return } (f s)$ )"

```

Modify the current state using the function passed in.

```

fun
  modify :: "('s  $\implies$  's)  $\implies$  (unit, 'e, 's) state_monad"
where "modify f = get  $\gg$  ( $\lambda s::'s. \text{put } (f s)$ )"

```

```

fun
  assert :: "'e  $\implies$  ('s  $\implies$  bool)  $\implies$  (unit, 'e, 's) state_monad" where
  "assert x t = ( $\lambda s. \text{if } (t s) \text{ then return } () \text{ s else throw } x s$ )"

```

```

fun
  option :: "'e  $\implies$  ('s  $\implies$  'a option)  $\implies$  ('a, 'e, 's) state_monad" where
  "option x f = ( $\lambda s. (\text{case } f s \text{ of }
    \text{Some } y \implies \text{return } y s
    | \text{None} \implies \text{throw } x s)$ )"

```

2.4.5 Some basic examples

```

lemma "do {
  x  $\leftarrow$  return 1;
  return (2::nat);
  return x
} =
  return 1  $\gg$  ( $\lambda x. \text{return } (2::nat) \gg (\lambda_. (\text{return } x))$ )" <proof>

```

```

lemma "do {
  x  $\leftarrow$  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  $\leftarrow$  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"

```

2.5 Hoare Logic (StateMonad)

named_theorems wprule

definition

```

valid :: "('s ⇒ bool) ⇒ ('a, 'e, 's) state_monad ⇒
  ('a ⇒ 's ⇒ bool) ⇒
  ('e ⇒ bool) ⇒ bool"
("_{P}/ _ /({Q}, {E})")

```

where

```

"_{P} f {Q}, {E} ≡ ∀s. P s → (case f s of
  Normal (r, s') ⇒ Q r s'
| Exception e ⇒ E e)"

```

lemma weaken:

```

assumes "{Q} f {R}, {E}"
and "∀s. P s → Q s"
shows "{P} f {R}, {E}"
⟨proof⟩

```

lemma strengthen:

```

assumes "{P} f {Q}, {E}"
and "∀a s. Q a s → R a s"
shows "{P} f {R}, {E}"
⟨proof⟩

```

definition wp

```

where "wp f P E s ≡ (case f s of
  Normal (r, s') ⇒ P r s'
| Exception e ⇒ E e)"

```

declare wp_def [solidity_symbex]

```

lemma wp_valid: assumes "∧s. P s ⇒ (wp f Q E s)" shows "{P} f {Q}, {E}"
⟨proof⟩

```

```

lemma valid_wp: assumes "{P} f {Q}, {E}" shows "∧s. P s ⇒ (wp f Q E s)"
⟨proof⟩

```

```

lemma put: "{λs. P () x} put x {P}, {E}"
⟨proof⟩

```

lemma put':

```

assumes "∀s. P s → Q () x"
shows "{λs. P s} put x {Q}, {E}"
⟨proof⟩

```

lemma wpput[wprule]:

```

assumes "P () x"

```

```

shows "wp (put x) P E s"
⟨proof⟩

lemma get: "⟦λs. P s s⟧ get ⟦P⟧, ⟦E⟧"
⟨proof⟩

lemma get':
  assumes "∀s. P s → Q s s"
  shows "⟦λs. P s⟧ get ⟦Q⟧, ⟦E⟧"
  ⟨proof⟩

lemma wpget[wprule]:
  assumes "P s s"
  shows "wp get P E s"
  ⟨proof⟩

lemma return: "⟦λs. P x s⟧ return x ⟦P⟧, ⟦E⟧"
⟨proof⟩

lemma return':
  assumes "∀s. P s → Q x s"
  shows "⟦λs. P s⟧ return x ⟦Q⟧, ⟦E⟧"
  ⟨proof⟩

lemma wpreturn[wprule]:
  assumes "P x s"
  shows "wp (return x) P E s"
  ⟨proof⟩

lemma bind:
  assumes "∀x. ⟦B x⟧ g x ⟦C⟧, ⟦E⟧"
  and "⟦A⟧ f ⟦B⟧, ⟦E⟧"
  shows "⟦A⟧ f ≧≧ g ⟦C⟧, ⟦E⟧"
  ⟨proof⟩

lemma wpbind[wprule]:
  assumes "wp f (λa. (wp (g a) P E)) E s"
  shows "wp (f ≧≧ g) P E s"
  ⟨proof⟩

lemma wpassert[wprule]:
  assumes "t s ⇒ wp (return ()) P E s"
  and "¬ t s ⇒ wp (throw x) P E s"
  shows "wp (assert x t) P E s"
  ⟨proof⟩

lemma throw:
  assumes "E x"
  shows "⟦P⟧ throw x ⟦Q⟧, ⟦E⟧"
  ⟨proof⟩

lemma wpthrow[wprule]:
  assumes "E x"
  shows "wp (throw x) P E s"
  ⟨proof⟩

lemma applyf:
  "⟦λs. P (f s) s⟧ applyf f ⟦λa s. P a s⟧, ⟦E⟧"
  ⟨proof⟩

lemma applyf':
  assumes "∀s. P s → Q (f s) s"
  shows "⟦λs. P s⟧ applyf f ⟦λa s. Q a s⟧, ⟦E⟧"
  ⟨proof⟩

```

2 Preliminaries

```
lemma wpapplyf[wprule]:
  assumes "P (f s) s"
  shows "wp (applyf f) P E s"
  <proof>

lemma modify:
  "{λs. P () (f s)} modify f {P}, {E}"
  <proof>

lemma modify':
  assumes "∀s. P s → Q () (f s)"
  shows "{λs. P s} modify f {Q}, {E}"
  <proof>

lemma wpmmodify[wprule]:
  assumes "P () (f s)"
  shows "wp (modify f) P E s"
  <proof>

lemma wpcasenat[wprule]:
  assumes "(y=(0::nat) ⇒ wp (f y) P E s)"
    and "∀x. y=Suc x ⇒ wp (g x) P E s"
  shows "wp (case y::nat of 0 ⇒ f y | Suc x ⇒ g x) P E s"
  <proof>

lemma wpif[wprule]:
  assumes "c ⇒ wp f P E s"
    and "¬c ⇒ wp g P E s"
  shows "wp (if c then f else g) P E s"
  <proof>

lemma wpsome[wprule]:
  assumes "∀y. x = Some y ⇒ wp (f y) P E s"
    and "x = None ⇒ wp g P E s"
  shows "wp (case x of Some y ⇒ f y | None ⇒ g) P E s"
  <proof>

lemma wption[wprule]:
  assumes "∀y. f s = Some y ⇒ wp (return y) P E s"
    and "f s = None ⇒ wp (throw x) P E s"
  shows "wp (option x f) P E s"
  <proof>

lemma wprod[wprule]:
  assumes "∀x y. a = (x,y) ⇒ wp (f x y) P E s"
  shows "wp (case a of (x, y) ⇒ f x y) P E s"
  <proof>

method wp = rule wprule; wp?
method wpcg = rule wp_valid, wp

lemma "{λs. s=5} do {
  put (5::nat);
  x ← get;
  return x
} {λa s. s=5}, {λe. False}"
  <proof>
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 :: "(nat ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ nat ⇒ 'b"
where
  "iter f v x = (if x ≤ 0 then v
                else f (x-1) (iter f v (x-1)))"

fun iter' :: "(nat ⇒ 'b ⇒ 'b option) ⇒ 'b ⇒ nat ⇒ '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

typedef bits = "{8::nat,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}"
  morphisms to_nat to_bit <proof>

setup_lifting type_definition_bits

lift_definition b8 :: "bits" is 8 <proof>
lift_definition b16 :: "bits" is 16 <proof>
lift_definition b24 :: "bits" is 24 <proof>
lift_definition b32 :: "bits" is 32 <proof>
lift_definition b40 :: "bits" is 40 <proof>
lift_definition b48 :: "bits" is 48 <proof>
lift_definition b56 :: "bits" is 56 <proof>
lift_definition b64 :: "bits" is 64 <proof>
lift_definition b72 :: "bits" is 72 <proof>
lift_definition b80 :: "bits" is 80 <proof>
lift_definition b88 :: "bits" is 88 <proof>
lift_definition b96 :: "bits" is 96 <proof>
lift_definition b104 :: "bits" is 104 <proof>
lift_definition b112 :: "bits" is 112 <proof>
lift_definition b120 :: "bits" is 120 <proof>
lift_definition b128 :: "bits" is 128 <proof>
lift_definition b136 :: "bits" is 136 <proof>
lift_definition b144 :: "bits" is 144 <proof>
lift_definition b152 :: "bits" is 152 <proof>
lift_definition b160 :: "bits" is 160 <proof>
lift_definition b168 :: "bits" is 168 <proof>
lift_definition b176 :: "bits" is 176 <proof>
lift_definition b184 :: "bits" is 184 <proof>
lift_definition b192 :: "bits" is 192 <proof>
lift_definition b200 :: "bits" is 200 <proof>
```

3 Types and Accounts

```
lift_definition b208 :: "bits" is 208 <proof>
lift_definition b216 :: "bits" is 216 <proof>
lift_definition b224 :: "bits" is 224 <proof>
lift_definition b232 :: "bits" is 232 <proof>
lift_definition b240 :: "bits" is 240 <proof>
lift_definition b248 :: "bits" is 248 <proof>
lift_definition b256 :: "bits" is 256 <proof>

declare b8.rep_eq[simp]
declare b16.rep_eq[simp]
declare b24.rep_eq[simp]
declare b32.rep_eq[simp]
declare b40.rep_eq[simp]
declare b48.rep_eq[simp]
declare b56.rep_eq[simp]
declare b64.rep_eq[simp]
declare b72.rep_eq[simp]
declare b80.rep_eq[simp]
declare b88.rep_eq[simp]
declare b96.rep_eq[simp]
declare b104.rep_eq[simp]
declare b112.rep_eq[simp]
declare b120.rep_eq[simp]
declare b128.rep_eq[simp]
declare b136.rep_eq[simp]
declare b144.rep_eq[simp]
declare b152.rep_eq[simp]
declare b160.rep_eq[simp]
declare b168.rep_eq[simp]
declare b176.rep_eq[simp]
declare b184.rep_eq[simp]
declare b192.rep_eq[simp]
declare b200.rep_eq[simp]
declare b208.rep_eq[simp]
declare b216.rep_eq[simp]
declare b224.rep_eq[simp]
declare b232.rep_eq[simp]
declare b240.rep_eq[simp]
declare b248.rep_eq[simp]
declare b256.rep_eq[simp]

instantiation bits :: linorder
begin
  lift_definition less_bits :: "bits  $\Rightarrow$  bits  $\Rightarrow$  bool" is "<" <proof>
  lift_definition less_eq_bits :: "bits  $\Rightarrow$  bits  $\Rightarrow$  bool" is " $\leq$ " <proof>
  instance <proof>
end

instantiation bits::equal
begin
  lift_definition equal_bits :: "bits  $\Rightarrow$  bits  $\Rightarrow$  bool" is "=" <proof>
  instance <proof>
end

declare Valuetypes.less_bits.rep_eq[simp]
declare Valuetypes.less_eq_bits.rep_eq[simp]

lemma to_nat_g_0[simp]: "to_nat b'>0" <proof>
lemma to_nat_max_g_0[simp]: "0 < max (to_nat b1) (to_nat b2)" <proof>
lemma to_nat_max[simp]: "max (to_nat b1) (to_nat b2) = to_nat (max b1 b2)" <proof>

datatype types = TSIInt bits
              | TUIInt bits
              | TBool
```

| TAddr

```

definition createSInt :: "bits  $\Rightarrow$  int  $\Rightarrow$  valuetype"
where
  "createSInt b v =
    (if v  $\geq$  0

      then ShowLint  $(-(2^{((to\_nat\ b)-1)}) + (v+2^{((to\_nat\ b)-1)}) \bmod (2^{(to\_nat\ b)}))$ 
      else ShowLint  $(2^{((to\_nat\ b)-1)} - (-v+2^{((to\_nat\ b)-1)-1}) \bmod (2^{(to\_nat\ b)} - 1))$ "

value "createSInt b16 (-002)"

declare createSInt_def [solidity_symbex]

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

lemma upper_bound2:
  fixes b::bits
  and c::int
  assumes "c < 2to_nat b"
  and "c  $\geq$  0"
  shows "c - (2((to_nat b)-1)) < 2((to_nat b)-1)"
  <proof>

lemma upper_bound3:
  fixes b::bits
  and v::int
  defines "x  $\equiv$  - (2((to_nat b) - 1)) + (v + 2((to_nat b) - 1)) mod 2(to_nat b)"
  shows "x < 2((to_nat b)-1)"
  <proof>

lemma upper_bound4:
  fixes b::bits
  and v::int
  assumes "to_nat(b) > 0"
  shows "2((to_nat(b))-1) - (-v+2((to_nat(b))-1)-1) mod (2to_nat(b)) - 1 < 2((to_nat(b)) - 1)"
  <proof>

lemma lower_bound:
  fixes b::bits
  shows " $\forall (c::int) \geq -(2^{((to\_nat\ b)-1)}). (-c + 2^{((to\_nat\ b)-1)} - 1 < 2^{(to\_nat\ b)})$ "
  <proof>

lemma lower_bound2:
  fixes b::bits
  and v::int
  defines "x  $\equiv$  2((to_nat b) - 1) - (-v+2((to_nat b)-1)-1) mod 2(to_nat b) - 1"
  shows "x  $\geq$  - (2((to_nat b) - 1))"
  <proof>

lemma createSInt_id_g0:
  fixes b::bits
  and v::int
  assumes "v  $\geq$  0"
  and "v < 2((to_nat b)-1)"
  shows "createSInt b v = ShowLint v"
  <proof>

```

```

lemma createSInt_id_l0:
  fixes b::bits
  and v::int
  assumes "v < 0"
  and "v ≥ -(2^(to_nat b)-1))"
  shows "createSInt b v = ShowLint v"
⟨proof⟩

lemma createSInt_id:
  fixes b::bits
  and v::int
  assumes "v < 2^(to_nat b)-1)"
  and "v ≥ -(2^(to_nat b)-1))"
  shows "createSInt b v = ShowLint v" ⟨proof⟩

definition createUInt :: "bits ⇒ int ⇒ valuetype"
  where "createUInt b v = ShowLint (v mod (2^(to_nat b))))"
declare createUInt_def [solidity_symbex]

lemma createSInt_less:
  shows "ReadLint (createSInt b v) < 2^(to_nat(b) - 1))"
⟨proof⟩

lemma createSInt_greater:
  shows "ReadLint (createSInt b v) ≥ - (2^(to_nat(b) - 1))"
⟨proof⟩

lemma "ShowLint (ReadLint STR ''001'') ≠ STR ''001''" ⟨proof⟩

definition checkSInt :: "bits ⇒ valuetype ⇒ bool"
  where
    "checkSInt b v = (ReadLint v ≥ -(2^(to_nat(b) - 1))) ∧ ReadLint v < 2^(to_nat(b) - 1)
      ∧ ShowLint (ReadLint v) = v)"
declare checkSInt_def [solidity_symbex]

lemma createUInt_greater:
  shows "ReadLint (createUInt b v) ≥ 0" ⟨proof⟩

lemma createUInt_less:
  shows "ReadLint (createUInt b v) < (2^(to_nat b)))" ⟨proof⟩

lemma createUInt_id:
  assumes "v ≥ 0"
  and "v < 2^(to_nat b)"
  shows "createUInt b v = ShowLint v"
⟨proof⟩

definition checkUInt :: "bits ⇒ valuetype ⇒ bool"
  where
    "checkUInt b v = (ReadLint v ≥ 0 ∧ ReadLint v < 2^(to_nat b) ∧ ShowLint (ReadLint v) = v)"
declare checkUInt_def [solidity_symbex]

definition createBool :: "bool ⇒ valuetype"
  where
    "createBool b = ShowLbool b"
declare createBool_def [solidity_symbex]

fun removeDot :: "char list ⇒ String.literal ⇒ String.literal"
  where
    "removeDot [] str = str"
  | "removeDot (x # xs) str = (if x = CHR '.' then removeDot xs str else removeDot xs (str +
    (String.implode [x])))"

```

```

lemma removeDotNoDot:
  assumes "CHR '.'' ∉ set(String.explode st)"
  and "1 = String.explode y"
  shows "(CHR '.'' ∉ set(String.explode(removeDot 1 st)))" <proof>

definition createAddress :: "address ⇒ valuetype"
where
  "createAddress ad = (if (CHR '.'' ∉ set(String.explode ad)) then ad else removeDot
(String.explode(ad)) (STR '''))"

lemma createAddressNoDots:
  shows "createAddress ad = t → CHR '.'' ∉ set(String.explode t)" <proof>

declare createAddress_def [solidity_symbex]

fun comp :: "types ⇒ types ⇒ bool"
where
  "comp (TSInt b1) (TSInt b2) = (to_nat b1 ≤ to_nat b2)"
| "comp (TUInt b1) (TUInt b2) = (to_nat b1 ≤ to_nat b2)"
| "comp (TUInt b1) (TSInt b2) = (to_nat b1 < to_nat b2)"
| "comp TBool TBool = True"
| "comp TAddr TAddr = True"
| "comp _ _ = False"

fun convert :: "types ⇒ types ⇒ valuetype ⇒ valuetype option"
where
  "convert (TSInt b1) (TSInt b2) v =
  (if to_nat b1 ≤ to_nat b2
  then Some v
  else None)"
| "convert (TUInt b1) (TUInt b2) v =
  (if to_nat b1 ≤ to_nat b2
  then Some v
  else None)"
| "convert (TUInt b1) (TSInt b2) v =
  (if to_nat b1 < to_nat b2
  then Some v
  else None)"
| "convert TBool TBool v = Some v"
| "convert TAddr TAddr v = Some v"
| "convert _ _ _ = None"

lemma convertSame:
  assumes "convert t1 t2 v = Some(v)"
  shows "v = v" <proof>

lemma convert_id[simp]:
  "convert tp tp kv = Some kv"
  <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 to_nat b2 < to_nat b1

```

3 Types and Accounts

```
    then Some (createSInt b1 (op [v1] [v2]), TSInt b1)
    else None)"
| "olift op (TUInt b1) (TSInt b2) v1 v2 =
    (if to_nat b1 < to_nat 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 to_nat b2 < to_nat b1
    then Some (createBool (op [v1] [v2]), TBool)
    else None)"
| "plift op (TUInt b1) (TSInt b2) v1 v2 =
    (if to_nat b1 < to_nat 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 (<)"

definition leq :: "types ⇒ types ⇒ valuetype ⇒ valuetype ⇒ (valuetype * types) option"
where
  "leq = plift (≤)"

declare add_def sub_def equal_def leq_def less_def [solidity_symbex]

fun vtand :: "types ⇒ types ⇒ valuetype ⇒ valuetype ⇒ (valuetype * types) option"
where
  "vtand TBool TBool a b =
    (if a = ShowLbool True ∧ b = ShowLbool True then Some (ShowLbool True, TBool)
    else Some (ShowLbool False, TBool))"
| "vtand _ _ _ _ = None"

fun vtor :: "types ⇒ types ⇒ valuetype ⇒ valuetype ⇒ (valuetype * types) option"
where
  "vtor TBool TBool a b =
    (if a = ShowLbool False ∧ b = ShowLbool False
```


3 Types and Accounts

```
definition emptyAccount :: "accounts"
where
  "emptyAccount _ = emptyAcc"

declare emptyAccount_def [solidity_symbex]

definition addBalance :: "address  $\Rightarrow$  valuetype  $\Rightarrow$  accounts  $\Rightarrow$  accounts option"
where
  "addBalance ad val acc =
    (if ReadLint val  $\geq$  0
     then (let v = ReadLint (Bal (acc ad)) + ReadLint val
          in if (v < 2256)
              then Some (acc(ad := acc ad (Bal:=ShowLint v)))
              else None)
     else None)"

declare addBalance_def [solidity_symbex]

lemma addBalance_val1:
  assumes "addBalance ad val acc = Some acc'"
  shows "ReadLint val  $\geq$  0"
  <proof>

lemma addBalance_val2:
  assumes "addBalance ad val acc = Some acc'"
  shows "ReadLint (Bal (acc ad)) + ReadLint val < 2256"
  <proof>

lemma addBalance_limit:
  assumes "addBalance ad val acc = Some acc'"
  and " $\forall$  ad. ReadLint (Bal (acc ad))  $\geq$  0  $\wedge$  ReadLint (Bal (acc ad)) < 2256"
  shows " $\forall$  ad. ReadLint (Bal (acc' ad))  $\geq$  0  $\wedge$  ReadLint (Bal (acc' ad)) < 2256"
  <proof>

lemma addBalance_add:
  assumes "addBalance ad val acc = Some acc'"
  shows "ReadLint (Bal (acc' ad)) = ReadLint (Bal (acc ad)) + ReadLint val"
  <proof>

lemma addBalance_mono:
  assumes "addBalance ad val acc = Some acc'"
  shows "ReadLint (Bal (acc' ad))  $\geq$  ReadLint (Bal (acc ad))"
  <proof>

lemma addBalance_eq:
  assumes "addBalance ad val acc = Some acc'"
  and "ad  $\neq$  ad'"
  shows "Bal (acc ad') = Bal (acc' ad)"
  <proof>

definition subBalance :: "address  $\Rightarrow$  valuetype  $\Rightarrow$  accounts  $\Rightarrow$  accounts option"
where
  "subBalance ad val acc =
    (if ReadLint val  $\geq$  0
     then (let v = ReadLint (Bal (acc ad)) - ReadLint val
          in if (v  $\geq$  0)
              then Some (acc(ad := acc ad (Bal:=ShowLint v)))
              else None)
     else None)"

declare subBalance_def [solidity_symbex]

lemma subBalance_val1:
```

```

    assumes "subBalance ad val acc = Some acc'"
    shows "ReadLint val ≥ 0"
⟨proof⟩

lemma subBalance_val2:
    assumes "subBalance ad val acc = Some acc'"
    shows "ReadLint (Bal (acc ad)) - ReadLint val ≥ 0"
⟨proof⟩

lemma subBalance_sub:
    assumes "subBalance ad val acc = Some acc'"
    shows "ReadLint (Bal (acc' ad)) = ReadLint (Bal (acc ad)) - ReadLint val"
⟨proof⟩

lemma subBalance_limit:
    assumes "subBalance ad val acc = Some acc'"
    and "∀ ad. ReadLint (Bal (acc ad)) ≥ 0 ∧ ReadLint (Bal (acc ad)) < 2 ^ 256"
    shows "∀ ad. ReadLint (Bal (acc' ad)) ≥ 0 ∧ ReadLint (Bal (acc' ad)) < 2 ^ 256"
⟨proof⟩

lemma subBalance_mono:
    assumes "subBalance ad val acc = Some acc'"
    shows "ReadLint (Bal (acc ad)) ≥ ReadLint (Bal (acc' ad))"
⟨proof⟩

lemma subBalance_eq:
    assumes "subBalance ad val acc = Some acc'"
    and "ad ≠ ad'"
    shows "(Bal (acc ad)) = (Bal (acc' ad'))"
⟨proof⟩

definition transfer :: "address ⇒ address ⇒ valuetype ⇒ accounts ⇒ accounts option"
where
    "transfer ads addr val acc =
      (case subBalance ads val acc of
        Some acc' ⇒ addBalance addr val acc'
      | None ⇒ None)"

declare transfer_def [solidity_symbex]

lemma transfer_val1:
    assumes "transfer ads addr val acc = Some acc'"
    shows "ReadLint val ≥ 0"
⟨proof⟩

lemma transfer_val2:
    assumes "transfer ads addr val acc = Some acc'"
    and "ads ≠ addr"
    shows "ReadLint (Bal (acc ads)) + ReadLint val < 2^256"
⟨proof⟩

lemma transfer_val3:
    assumes "transfer ads addr val acc = Some acc'"
    shows "ReadLint (Bal (acc ads)) - ReadLint val ≥ 0"
⟨proof⟩

lemma transfer_add:
    assumes "transfer ads addr val acc = Some acc'"
    and "addr ≠ ads"
    shows "ReadLint (Bal (acc' addr)) = ReadLint (Bal (acc ads)) + ReadLint val"
⟨proof⟩

lemma transfer_sub:
    assumes "transfer ads addr val acc = Some acc'"

```

3 Types and Accounts

```
    and "addr ≠ ads"
    shows "ReadLint (Bal (acc' ads)) = ReadLint (Bal (acc ads)) - ReadLint val"
⟨proof⟩
```

```
lemma transfer_same:
  assumes "transfer ad ad' val acc = Some acc'"
  and "ad = ad'"
  shows "ReadLint (Bal (acc ad)) = ReadLint (Bal (acc' ad))"
⟨proof⟩
```

```
lemma transfer_mono:
  assumes "transfer ads addr val acc = Some acc'"
  shows "ReadLint (Bal (acc' addr)) ≥ ReadLint (Bal (acc addr))"
⟨proof⟩
```

```
lemma transfer_eq:
  assumes "transfer ads addr val acc = Some acc'"
  and "ad ≠ ads"
  and "ad ≠ addr"
  shows "Bal (acc' ad) = Bal (acc ad)"
⟨proof⟩
```

```
lemma transfer_limit:
  assumes "transfer ads addr val acc = Some acc'"
  and "∀ ad. ReadLint (Bal (acc ad)) ≥ 0 ∧ ReadLint (Bal (acc ad)) < 2 ^ 256"
  shows "∀ ad. ReadLint (Bal (acc' ad)) ≥ 0 ∧ ReadLint (Bal (acc' ad)) < 2 ^ 256"
⟨proof⟩
```

```
lemma transfer_type_same:
  assumes "transfer ads addr val acc = Some acc'"
  shows "Type (acc' ad) = Type (acc ad)"
⟨proof⟩
```

```
lemma transfer_contracts_same:
  assumes "transfer ads addr val acc = Some acc'"
  shows "Contracts (acc' ad) = Contracts (acc ad)"
⟨proof⟩
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 "Finite-Map-Extras.Finite_Map_Extras"
```

```
begin
```

4.1.1 Hashing

```
definition hash :: "location  $\Rightarrow$  String.literal  $\Rightarrow$  location"
  where "hash loc ix = ix + (STR '.' + loc)"
```

```
declare hash_def [solidity_symbex]
```

Hash function which uses - for contract version/instance number

```
definition hash_version :: "location  $\Rightarrow$  String.literal  $\Rightarrow$  location"
  where "hash_version loc ix = ix + (STR '-' + loc)"
```

```
lemma explode_dot[simp]: "literal.explode STR '.' = [CHR '.']"
  <proof>
```

```
lemma example: "hash (STR '1.0') (STR '2') = hash (STR '0') (STR '2.1')" <proof>
```

```
lemma hash_explode:
  "String.explode (hash l i) = String.explode i @ (String.explode (STR '.')) @ String.explode l"
  <proof>
```

```
lemma hash_dot:
  "String.explode (hash l i) ! length (String.explode i) = CHR '.'"
  <proof>
```

```
lemma hash_injective:
  assumes "hash l i = hash l' i'"
  and "CHR '.'  $\notin$  set (String.explode i)"
  and "CHR '.'  $\notin$  set (String.explode i)"
  shows "l = l'  $\wedge$  i = i'"
  <proof>
```

4.1.2 General store

```
record 'v store =
  Mapping :: "(location, 'v) fmap"
  Toploc :: nat
```

```
definition accessStore :: "location  $\Rightarrow$  ('a, 'v) store_scheme  $\Rightarrow$  'a option"
  where "accessStore loc st = fmllookup (Mapping st) loc"
```

```
declare accessStore_def[solidity_symbex]
```

```
definition emptyStore :: "'v store"
  where "emptyStore = (| Mapping=fmempty, Toploc=0 |)"
```

```

declare emptyStore_def [solidity_symbex]

definition allocate :: "('a, 'v) store_scheme ⇒ location * ('a, 'v) store_scheme"
where "allocate s = (let ntop = Suc(Toploc s) in (ShowLnat ntop, s (|Toploc := ntop|)))"

lemma allocateMapping:
  assumes "∃t. (t, s') = allocate s"
  shows "Mapping s = Mapping s'" <proof>

lemma accessAllocate:
  assumes "∃t. (t, s') = allocate s"
  shows "∀l. accessStore l s' = accessStore l s"
  <proof>

lemma allocateSameAccess:
  shows "∀loc. accessStore loc m = accessStore loc (snd (allocate m))"
  <proof>

definition updateStore :: "location ⇒ 'v ⇒ ('v, 'a) store_scheme ⇒ ('v, 'a) store_scheme"
where "updateStore loc val s = s (| Mapping := fmupd loc val (Mapping s)|)"

declare updateStore_def [solidity_symbex]

lemma accessStore_updateStore[simp]: "accessStore l (updateStore l v st) = Some v" <proof>

lemma accessStore_non_changed:
  assumes "updateStore l v st = st'"
  shows "∀l'. l' ≠ l → accessStore l' st = accessStore l' st'"
  <proof>

definition push :: "'v ⇒ ('v, 'a) store_scheme ⇒ ('v, 'a) store_scheme"
  where "push val sto = (let s = updateStore (ShowLnat (Toploc sto)) val sto in snd (allocate s))"

declare push_def [solidity_symbex]

```

4.1.3 stack

```

datatype (discs_sels) stackvalue = KValue valuetype
  | KCDptr location
  | KMemptr location
  | KStoptr location

```

```

type_synonym stack = "stackvalue store"

```

4.1.4 Storage

Definition

```

type_synonym storagevalue = valuetype

type_synonym storageT = "(location, storagevalue) fmap"

datatype (discs_sels) stypes
  = STArray nat stypes
  | STMap types stypes
  | STValue types

```

Example

```

abbreviation mystorage :: storageT
where "mystorage ≡ (fmap_of_list
  [(STR ''0.0.0'', STR ''False''),

```

```
(STR ''1.1.0'', STR ''True'')])"
```

Access storage

```
definition accessStorage :: "types  $\Rightarrow$  location  $\Rightarrow$  storageT  $\Rightarrow$  storagevalue"
where
```

```
"accessStorage t loc sto =
  (case sto $$ loc of
    Some v  $\Rightarrow$  v
  | None  $\Rightarrow$  ival t)"
```

```
declare accessStorage_def [solidity_symbex]
```

Copy from storage to storage

```
primrec copyRec :: "location  $\Rightarrow$  location  $\Rightarrow$  stypes  $\Rightarrow$  storageT  $\Rightarrow$  storageT option"
where
```

```
"copyRec ls ld (STArray x t) sto =
  iter' ( $\lambda$ i s'. copyRec (hash ls (ShowLnat i)) (hash ld (ShowLnat i)) t s') sto x"
| "copyRec ls ld (STValue t) sto =
  (let e = accessStorage t ls sto in Some (fmupd ld e sto))"
| "copyRec _ _ (STMap _ _) _ = None"
```

```
definition copy :: "location  $\Rightarrow$  location  $\Rightarrow$  nat  $\Rightarrow$  stypes  $\Rightarrow$  storageT  $\Rightarrow$  storageT option"
where
```

```
"copy ls ld x t sto =
  iter' ( $\lambda$ i s'. copyRec (hash ls (ShowLnat i)) (hash ld (ShowLnat i)) t s') sto x"
```

```
declare copy_def [solidity_symbex]
```

```
abbreviation mystorage2::storageT
```

```
where "mystorage2  $\equiv$  (fmap_of_list
  [(STR ''0.0.0'', STR ''False''),
   (STR ''1.1.0'', STR ''True''),
   (STR ''0.5'', STR ''False''),
   (STR ''1.5'', STR ''True'')])"
```

```
lemma "copy (STR ''1.0'') (STR ''5'') 2 (STValue TBool) mystorage = Some mystorage2"
  <proof>
```

4.1.5 Memory and Calldata

Definition

```
datatype memoryvalue =
  MValue valuetype
  | MPointer location
```

```
datatype (discs_sels) mtypes =
  MArray nat mtypes
  | MValue types
```

4.1.6 Typed Store

```
record ('t, 'v) typedstore = "'v store" +
  Typed_Mapping :: "(location, 't) fmap"
```

Use the inherited accessStore for values, add new functions for type management

```
definition accessTypeStore :: "location  $\Rightarrow$  ('t, 'v) typedstore  $\Rightarrow$  't option"
where "accessTypeStore loc st = fmlookup (Typed_Mapping st) loc"
```

```
declare accessTypeStore_def [solidity_symbex]
```

```
definition updateTypeStore :: "location  $\Rightarrow$  't  $\Rightarrow$  ('t, 'v) typedstore  $\Rightarrow$  ('t, 'v) typedstore"
```

4 Stores and Environment

```
where "updateTypeStore loc typ st = st (| Typed_Mapping := fmupd loc typ (Typed_Mapping st) |)"

declare updateTypeStore_def [solidity_symbex]

definition updateTypedStore :: "location  $\Rightarrow$  'v  $\Rightarrow$  't  $\Rightarrow$  ('t, 'v) typedstore  $\Rightarrow$  ('t, 'v) typedstore"
where "updateTypedStore loc val typ st = updateTypeStore loc typ (updateStore loc val st)"

definition emptyTypedStore :: "('t, 'v) typedstore"
  where "emptyTypedStore = (| Mapping=fmempty, Toploc=0, Typed_Mapping=fmempty |)"

declare emptyTypedStore_def [solidity_symbex]

definition pushTyped :: "'v  $\Rightarrow$  't  $\Rightarrow$  ('t, 'v) typedstore  $\Rightarrow$  ('t, 'v) typedstore"
  where "pushTyped val typ sto = (let s = updateTypedStore (ShowLnat (Toploc sto)) val typ sto in snd
(allocate s))"

declare pushTyped_def [solidity_symbex]

lemma "accessStore l (updateTypedStore l v t (emptyTypedStore)) = Some v"
  <proof>

lemma "accessTypeStore l (updateTypedStore l v t (emptyTypedStore)) = Some t"
  <proof>

lemma allocateTypeSameAccess:
  shows " $\forall$  loc. accessTypeStore loc m = accessTypeStore loc (snd (allocate m))"
  <proof>

type_synonym memoryT = "(mtypes, memoryvalue) typedstore"

type_synonym calldataT = memoryT
```

Example

```
abbreviation mymemory::memoryT
  where "mymemory  $\equiv$ 
  (|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,
  Typed_Mapping = fmap_of_list
  [(STR ''1.1.0'', MTValue TBool),
  (STR ''0.1.0'', MTValue TBool),
  (STR ''1.0'', MTArray 2 (MTValue TBool)),
  (STR ''1.0.0'', MTValue TBool),
  (STR ''0.0.0'', MTValue TBool),
  (STR ''0.0'', MTArray 2 (MTValue TBool))])"
```

Initialization

Definition

```
primrec minitRec :: "location  $\Rightarrow$  mtypes  $\Rightarrow$  memoryT  $\Rightarrow$  memoryT"
where
  "minitRec loc (MTArray x t) = ( $\lambda$ mem.
  let m = updateTypedStore loc (MPointer loc) (MTArray x t) mem
  in iter ( $\lambda$ i m' . minitRec (hash loc (ShowLnat i)) t m') m x)"
| "minitRec loc (MValue t) = updateTypedStore loc (MValue (ival t)) (MValue t)"

definition minit :: "nat  $\Rightarrow$  mtypes  $\Rightarrow$  memoryT  $\Rightarrow$  memoryT"
where
```

```
"minit x t mem =
  (let l = ShowLnat (Toploc mem);
    m = iter (λi m' . minitRec (hash l (ShowLnat i)) t m') mem x
    in snd (allocate m))"
```

```
declare minit_def [solidity_symbex]
```

```
primrec arraysGreaterZero::"mtypes ⇒ bool"
  where "arraysGreaterZero (MTValue v) = True"
        | "arraysGreaterZero (MTArray x t) = (x>0 ∧ arraysGreaterZero t)"
```

Example

```
lemma "minit 2 (MTArray 2 (MTValue TBool)) emptyTypedStore =
  (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,
    Typed_Mapping = fmap_of_list
    [(STR ''0.0'', MTArray 2 (MTValue TBool)),
     (STR ''0.0.0'', MTValue TBool),
     (STR ''1.0.0'', MTValue TBool),
     (STR ''1.0'', MTArray 2 (MTValue TBool)),
     (STR ''0.1.0'', MTValue TBool),
     (STR ''1.1.0'', MTValue TBool)])" <proof>
```

Copy from memory to memory

Definition

```
primrec cpm2mrec :: "location ⇒ location ⇒ mtypes ⇒ memoryT ⇒ memoryT ⇒ memoryT option"
  where
    "cpm2mrec ls ld (MTArray x t) ms md =
      (case accessStore ls ms of
        Some (MPointer l) ⇒
          (let m = updateTypedStore ld (MPointer ld) (MTArray x t) md
            in iter' (λi m' . cpm2mrec (hash l (ShowLnat i)) (hash ld (ShowLnat i)) t ms m') m x)
          | _ ⇒ None)"
    | "cpm2mrec ls ld (MTValue t) ms md =
      (case accessStore ls ms of
        Some (MValue v) ⇒ Some (updateTypedStore ld (MValue v) (MTValue t) md)
        | _ ⇒ None)"
```

```
definition cpm2m :: "location ⇒ location ⇒ nat ⇒ mtypes ⇒ memoryT ⇒ memoryT ⇒ memoryT option"
  where
```

```
"cpm2m ls ld x t ms md = iter' (λi m . cpm2mrec (hash ls (ShowLnat i)) (hash ld (ShowLnat i)) t ms m)
  md x"
```

```
declare cpm2m_def [solidity_symbex]
```

Example

```
lemma "cpm2m (STR ''0'') (STR ''0'') 2 (MTArray 2 (MTValue TBool)) mymemory (snd (allocate
  emptyTypedStore)) = Some mymemory"
  <proof>
```

```
abbreviation mymemory2::memoryT
```

```
  where "mymemory2 ≡
    (Mapping = fmap_of_list
      [(STR ''0.5'', MValue STR ''True''),
       (STR ''1.5'', MValue STR ''False'')],
```

4 Stores and Environment

```
Toploc = 0,  
Typed_Mapping = fmap_of_list  
  [(STR ''0.5'', MValue TBool),  
   (STR ''1.5'', MValue TBool)]])"
```

```
lemma "cpm2m (STR ''1.0'') (STR ''5'') 2 (MValue TBool) mymemory emptyTypedStore = Some mymemory2"  
<proof>
```

4.1.7 Copy from storage to memory

Definition

```
fun cps2mTypeConvert :: "stypes  $\Rightarrow$  mtypes option"  
  where  
    "cps2mTypeConvert (STValue t) = Some (MValue t)"  
  | "cps2mTypeConvert (STArray len t) = (case (cps2mTypeConvert t) of Some t'  $\Rightarrow$  Some (MArray len t')  
                                           | None  $\Rightarrow$  None)"  
  | "cps2mTypeConvert _ = None"
```

```
primrec cps2mrec :: "location  $\Rightarrow$  location  $\Rightarrow$  stypes  $\Rightarrow$  storageT  $\Rightarrow$  memoryT  $\Rightarrow$  memoryT option"
```

where

```
"cps2mrec locs locm (STArray x t) sto mem =  
  (case cps2mTypeConvert t of  
    Some t'  $\Rightarrow$   
      (let m = updateTypedStore locm (MPointer locm) (MArray x t') mem  
          in iter' ( $\lambda$ i m'. cps2mrec (hash locs (ShowLnat i)) (hash locm (ShowLnat i)) t sto m') m x)  
    | None  $\Rightarrow$  None)"  
| "cps2mrec locs locm (STValue t) sto mem =  
  (let v = accessStorage t locs sto  
      in Some (updateTypedStore locm (MValue v) (MValue t) mem))"  
| "cps2mrec _ _ (STMap _ _) _ _ = None"
```

```
definition cps2m :: "location  $\Rightarrow$  location  $\Rightarrow$  nat  $\Rightarrow$  stypes  $\Rightarrow$  storageT  $\Rightarrow$  memoryT  $\Rightarrow$  memoryT option"
```

where

```
"cps2m locs locm x t sto mem =  
  iter' ( $\lambda$ i m'. cps2mrec (hash locs (ShowLnat i)) (hash locm (ShowLnat i)) t sto m') mem x"
```

```
declare cps2m_def [solidity_symbex]
```

Example

```
abbreviation mystorage3 :: storageT
```

```
where "mystorage3  $\equiv$  (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'')])"
```

```
lemma "cps2m (STR ''1'') (STR ''0'') 2 (STArray 2 (STValue TBool)) mystorage3 (snd (allocate  
emptyTypedStore)) = Some mymemory"  
<proof>
```

```
fun cps2mTypeCompatible :: "stypes  $\Rightarrow$  mtypes  $\Rightarrow$  bool"
```

where

```
"cps2mTypeCompatible (STValue t) (MValue t') = (t = t')"  
| "cps2mTypeCompatible (STArray len t) (MArray len' t') = (len' > 0  $\wedge$  len = len'  $\wedge$  cps2mTypeCompatible  
t t')"  
| "cps2mTypeCompatible _ _ = False"
```

4.1.8 Copy from memory to storage

Definition

```
primrec cpm2srec :: "location  $\Rightarrow$  location  $\Rightarrow$  mtypes  $\Rightarrow$  memoryT  $\Rightarrow$  storageT  $\Rightarrow$  storageT option"
```

where

```

"cpm2srec locm locs (MArray x t) mem sto =
  (case accessStore locm mem of
    Some (MPointer l) =>
      iter' (λi s'. cpm2srec (hash l (ShowLnat i)) (hash locs (ShowLnat i)) t mem s') sto x
    | _ => None)"
| "cpm2srec locm locs (MValue t) mem sto =
  (case accessStore locm mem of
    Some (MValue v) => Some (fmupd locs v sto)
    | _ => None)"

definition cpm2s :: "location => location => nat => mtypes => memoryT => storageT => storageT option"
where
  "cpm2s locm locs x t mem sto =
    iter' (λi s'. cpm2srec (hash locm (ShowLnat i)) (hash locs (ShowLnat i)) t mem s') sto x"

declare cpm2s_def [solidity_symbex]

```

Example

```

lemma "cpm2s (STR ''0'') (STR ''1'') 2 (MArray 2 (MValue TBool)) mymemory fmempty = Some mystorage3"
  <proof>

declare copyRec.simps [simp del, solidity_symbex add]
declare minitRec.simps [simp del, solidity_symbex add]
declare cpm2mrec.simps [simp del, solidity_symbex add]
declare cps2mrec.simps [simp del, solidity_symbex add]
declare cpm2srec.simps [simp del, solidity_symbex add]

end

```

4.2 Environment and state (Environment)

```

theory Environment
imports Accounts Storage StateMonad
begin

```

4.2.1 Environment

```

datatype (discs_sels) type
  = Value types
  | Calldata mtypes
  | Memory mtypes
  | Storage stypes

datatype denvalue = Stackloc location
                  | Storeloc location

record environment =
  Address :: address
  Contract :: Identifier
  Sender :: address
  Svalue :: valuetype
  Denvalue :: "(Identifier, type × denvalue) fmap"

fun identifiers :: "environment => Identifier fset"
  where "identifiers e = fmdom (Denvalue e)"

definition emptyEnv :: "address => Identifier => address => valuetype => environment"
  where "emptyEnv a c s v = (Address = a, Contract = c, Sender = s, Svalue = v, Denvalue = fmempty)"

declare emptyEnv_def [solidity_symbex]

lemma emptyEnv_address[simp]:
  "Address (emptyEnv a c s v) = a"

```

4 Stores and Environment

<proof>

lemma *emptyEnv_members[simp]*:
 "Contract (emptyEnv a c s v) = c"
<proof>

lemma *emptyEnv_sender[simp]*:
 "Sender (emptyEnv a c s v) = s"
<proof>

lemma *emptyEnv_svalue[simp]*:
 "Svalue (emptyEnv a c s v) = v"
<proof>

lemma *emptyEnv_denvalue[simp]*:
 "Denvalue (emptyEnv a c s v) = {\$\$}"
<proof>

definition *eempty* :: "environment"
 where "eempty = emptyEnv (STR ''') (STR ''') (STR ''') (STR ''')"

declare *eempty_def* [solidity_symbex]

fun *updateEnv* :: "Identifier \Rightarrow type \Rightarrow denvalue \Rightarrow environment \Rightarrow environment"
 where "updateEnv i t v e = e (Denvalue := fmupd i (t,v) (Denvalue e))"

fun *updateEnvOption* :: "Identifier \Rightarrow type \Rightarrow denvalue \Rightarrow environment \Rightarrow environment option"
 where "updateEnvOption i t v e = (case Denvalue e \$\$ i of
 Some _ \Rightarrow None
 | None \Rightarrow Some (updateEnv i t v e))"

lemma *updateEnvOption_address*: "updateEnvOption i t v e = Some e' \implies Address e = Address e'"
<proof>

fun *updateEnvDup* :: "Identifier \Rightarrow type \Rightarrow denvalue \Rightarrow environment \Rightarrow environment"
 where "updateEnvDup i t v e = (case Denvalue e \$\$ i of
 Some _ \Rightarrow e
 | None \Rightarrow 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_contract[simp]*: "Contract (updateEnvDup i t v e) = Contract e"
<proof>

lemma *updateEnv_den_sub*: "fndom (Denvalue e) \sqsubseteq fndom(Denvalue (updateEnv i t v e))" *<proof>*

lemma *updateEnvDup_dup*:
 assumes "i \neq i'" shows "fmlookup (Denvalue (updateEnvDup i t v e)) i' = Denvalue e \$\$ i'"
<proof>

lemma *env_reorder_neq*:
 assumes "x \neq 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 (ShowLnat (Toploc s))) e))"

fun astack_dup :: "Identifier ⇒ type ⇒ stackvalue ⇒ stack * environment ⇒ stack * environment"
where "astack_dup i t v (s, e) =
      (case fmlookup (Denvalue e) i of
       Some _ ⇒ (s, e)
       | None ⇒ (push v s, updateEnv i t (Stackloc (ShowLnat (Toploc
s))) e))
"
lemma astack_dup_is_astack:
assumes "Denvalue e $$ i = None"
shows "astack i t v (s, e) = astack_dup i t v (s, e)" ⟨proof⟩

lemma astack_den_sub:"fmdom (Denvalue e) |⊆| fmdom(Denvalue (snd (astack i t v (s, e))))"
⟨proof⟩

lemma astack_dup_den_sub:"fmdom (Denvalue e) |⊆| fmdom(Denvalue (snd (astack_dup i t v (s, e))))"
⟨proof⟩

lemma astack_dup_env:
assumes "astack_dup i t v (s,e) = (sk, env)"
shows "Address e = Address env ∧ Contract e = Contract env ∧ Sender e = Sender env ∧ Svalue e =
Svalue env"
⟨proof⟩

lemma astack_dup_denvalue:
assumes "astack_dup i t v (s,e) = (sk, env)"
shows "Denvalue e $$ i = Some y ⟶ Denvalue env $$ i = Some y"
⟨proof⟩

```

Examples

```

abbreviation "myenv::environment ≡ eempty (Denvalue := fmupd STR ''id1'' (Value TBool, Stackloc STR
''0'') fmemory)"
abbreviation "mystack::stack ≡ (Mapping = fmupd (STR ''0'') (KValue STR ''True'') fmemory, Toploc = 1)"

abbreviation "myenv2::environment ≡ eempty (Denvalue := fmupd STR ''id2'' (Value TBool, Stackloc STR
''1'') (fmupd STR ''id1'' (Value TBool, Stackloc STR ''0'') fmemory))"
abbreviation "mystack2::stack ≡ (Mapping = fmupd (STR ''1'') (KValue STR ''False'') (fmupd (STR ''0'')
(KValue STR ''True'') fmemory), Toploc = 2)"

lemma "astack (STR ''id1'') (Value TBool) (KValue (STR ''True'')) (emptyStore, eempty) =
(mystack,myenv)" ⟨proof⟩
lemma "astack (STR ''id2'') (Value TBool) (KValue (STR ''False'')) (mystack, myenv) =
(mystack2,myenv2)" ⟨proof⟩

```

4.2.2 Declarations

This function is used to declare a new variable: `decl id tp val copy cd mem sto c m k e`

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

sto is the original storage which is used as a source

c is the new calldata which is updated

m is the new memory which is updated

k is the new calldata which is updated

e is the new environment which is updated

```

fun decl :: "Identifier => type => (stackvalue * type) option => bool => calldataT => memoryT =>
(storageT)
=> calldataT x memoryT x stack x environment => (calldataT x memoryT x stack x environment)
option"
  where

    "decl i (Value t) None _ _ _ _ (c, m, k, e) =

      Some (c, m, (astack_dup i (Value t) (KValue (ival t)) (k, e)))
    "
  | "decl i (Value t) (Some (KValue v, Value t')) _ _ _ _ (c, m, k, e) =
    Option.bind (convert t' t v)
      (\v'. Some (c, m, astack_dup i (Value t) (KValue v') (k, e)))"
  | "decl _ (Value _) _ _ _ _ _ = None"

  | "decl i (Callldata (MTArray x t)) (Some (KCDptr p, t')) True cd _ _ (c, m, k, e) =
    (if (t' = (Callldata (MTArray x t)) ^ x > 0 ^ Denvalue e $$ i = None) then
      (let l = ShowLnat (Toploc c);
        (_, c') = allocate c
        in Option.bind (cpm2m p l x t cd c')
          (\c''. Some (c'', m, astack_dup i (Callldata (MTArray x t)) (KCDptr l) (k, e)))) else None)"
  | "decl i (Callldata (MTArray x t)) (Some (KMemptr p, t')) True _ mem _ (c, m, k, e) =
    (if (t' = (Memory (MTArray x t)) ^ x > 0 ^ Denvalue e $$ i = None) then
      (let l = ShowLnat (Toploc c);
        (_, c') = allocate c
        in Option.bind (cpm2m p l x t mem c')
          (\c''. Some (c'', m, astack_dup i (Callldata (MTArray x t)) (KCDptr l) (k, e)))) else None)"
  | "decl i (Callldata _) _ _ _ _ _ = None"

  | "decl i (Memory (MTArray x t)) None _ _ _ _ (c, m, k, e) =
    (if Denvalue e $$ i = None ^ arraysGreaterZero (MTArray x t) then
      (let m' = minit x t m
        in Some (c, m', astack_dup i (Memory (MTArray x t)) (KMemptr (ShowLnat (Toploc m))) (k, e))
      )else None)"
  | "decl i (Memory (MTArray x t)) (Some (KMemptr p, t')) True _ mem _ (c, m, k, e) =
    (if (t' = (Memory (MTArray x t)) ^ x > 0 ^ Denvalue e $$ i = None) then
      Option.bind (cpm2m p (ShowLnat (Toploc m)) x t mem (snd (allocate m)))
        (\m'. Some (c, m', astack_dup i (Memory (MTArray x t)) (KMemptr (ShowLnat (Toploc m))) (k, e))) else
      None)"
  | "decl i (Memory (MTArray x t)) (Some (KMemptr p, t')) False _ _ _ (c, m, k, e) =
    (if (t' = (Memory (MTArray x t)) ^ x > 0 ^ Denvalue e $$ i = None) then
      Some (c, m, astack_dup i (Memory (MTArray x t)) (KMemptr p) (k, e)) else None)"
  | "decl i (Memory (MTArray x t)) (Some (KCDptr p, t')) _ cd _ _ (c, m, k, e) =
    (if (t' = (Callldata (MTArray x t)) ^ x > 0 ^ Denvalue e $$ i = None) then
      Option.bind (cpm2m p (ShowLnat (Toploc m)) x t cd (snd (allocate m)))
    "

```

```

  (λm'. Some (c, m', astack_dup i (Memory (MArray x t)) (KMemptr (ShowLnat (Toploc m)))) (k, e))) else
None)"
| "decl i (Memory (MArray x t)) (Some (KStoptr p, Storage (STArray x' t'))) _ _ _ s (c, m, k, e) =
  (if (cps2mTypeCompatible (STArray x' t') (MArray x t) ∧ Denvalue e $$ i = None) then
    Option.bind (cps2m p (ShowLnat (Toploc m))) x' t' s (snd (allocate m)))
    (λm''. Some (c, m'', astack_dup i (Memory (MArray x t)) (KMemptr (ShowLnat (Toploc m)))) (k, e)))
else None)"
| "decl _ (Memory _) _ _ _ _ _ = None"

```

```

| "decl i (Storage (STArray x t)) (Some (KStoptr p, t')) _ _ _ _ (c, m, k, e) =
  (if (t' = (Storage (STArray x t)) ∧ Denvalue e $$ i = None) then Some (c, m, astack_dup i (Storage
(STArray x t)) (KStoptr p) (k, e)) else None)"
| "decl i (Storage (STMap t t')) (Some (KStoptr p, t')) _ _ _ _ (c, m, k, e) =
  (if (t'' = (Storage (STMap t t')) ∧ Denvalue e $$ i = None) then Some (c, m, astack_dup i (Storage
(STMap t t')) (KStoptr p) (k, e)) else None)"
| "decl _ (Storage _) _ _ _ _ _ = None"

```

lemma decl_env:

```

  assumes "decl a1 a2 a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
  shows "Address env = Address env' ∧ Sender env = Sender env' ∧ Svalue env = Svalue env' ∧ Contract
env = Contract env' "
  ⟨proof⟩

```

lemma decl_env_subset_denval:

```

  assumes "decl a1 a2 a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
  shows "fmdom (Denvalue env) |⊆| fmdom (Denvalue env')"
  "
  ⟨proof⟩

```

lemma decl_env_kval:

```

  assumes "decl i (Value t) a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env'"
  and "Denvalue env $$ i = None"
  shows "∃ t l. Denvalue env' $$ i = Some (Value t, Stackloc l)"
  ⟨proof⟩

```

lemma decl_env_memory:

```

  assumes "decl i (Memory t) a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env'"
  and "Denvalue env $$ i = None"
  shows "Denvalue env' $$ i = Some (Memory t, Stackloc (ShowLnat (Toploc k)))
  ∧ (accessStore (ShowLnat (Toploc k)) k' = Some (KMemptr (ShowLnat (Toploc m))) ∨ (∃ v t'. a3 =
Some(KMemptr v, Memory t') ∧ accessStore (ShowLnat (Toploc k)) k' = Some (KMemptr v)))"
  ⟨proof⟩

```

lemma decl_env_storage:

```

  assumes "decl i tp a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env'"
  and "∀ stp. tp ≠ Storage stp"
  and "Denvalue env $$ i = None"
  shows "∀ t l. Denvalue env' $$ i = Some (t, l) → (∀ stl tp. l ≠ Storeloc stl ∧ t ≠ Storage tp)"
  ⟨proof⟩

```

lemma decl_stack_change:

```

  assumes "decl a1 a2 a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env'"
  and "l ≠ (ShowLnat (Toploc k))"
  shows "(accessStore l k = accessStore l k'"
  ⟨proof⟩

```

lemma decl_KValue_tp_match:

```

  assumes "decl i tp (Some (a1, Value t)) cp cd mem sto (c, m, k, env) = Some (c', m', k', env'"
  shows "∃ vv. tp = Value vv" ⟨proof⟩

```

4 Stores and Environment

lemma decl_Calldata_tp_match:

assumes "decl i tp (Some (a1, Calldata t)) cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 shows " $\exists vv. tp = Calldata vv \vee tp = Memory vv$ " *<proof>*

lemma decl_Memory_tp_match:

assumes "decl i tp (Some (a1, Memory t)) cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 shows " $\exists vv. tp = Calldata vv \vee tp = Memory vv$ " *<proof>*

lemma decl_Memory_tp_options:

assumes "decl i tp (Some (a1, Storage t)) cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 shows " $(\exists x t' t''. t = STArray x t' \wedge cps2mTypeCompatible (STArray x t') (MTArray x t'')) \wedge tp = Memory (MTArray x t'')$ "
<proof>

lemma decl_StorageStack_options:

assumes "decl i tp a1 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 and " $\forall x. tp \neq Storage x$ "
 and "(Denvalue env') \$\$ ip' = Some ((Storage t, Stackloc l))"
 and "accessStore l k' = Some (KStoptr ptr)"
 shows "accessStore l k = accessStore l k'"
<proof>

lemma decl_env_storlocs_unchanged:

assumes "decl i tp a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 shows " $\forall t l. Denvalue env' $$ i = Some (t, Storeloc l) \longrightarrow Denvalue env $$ i = Some (t, Storeloc l)$ "
<proof>

lemma decl_env_monotonic:

assumes "decl i tp a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 and "Denvalue env \$\$ ii = Some x"
 shows "Denvalue env' \$\$ ii = Some x"
<proof>

lemma decl_env_not_i:

assumes "decl i tp a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 and "Denvalue env' \$\$ ii = Some x"
 and " $ii \neq i$ "
 shows "Denvalue env \$\$ ii = Some x"
<proof>

lemma decl_some_same:

assumes "decl i tp a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 and "Denvalue env \$\$ i = Some x"
 shows " $env' = env \wedge c = c' \wedge m = m' \wedge k = k'$ "
<proof>

lemma decl_storage_tp:

assumes "decl i tp a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 and "Denvalue env \$\$ i = None"
 and "Denvalue env' \$\$ i = Some (Storage tt, Stackloc (ShowL_{nat} (Toploc k)))"
 shows " $\exists p. a3 = (Some (KStoptr p, Storage tt))$ "
<proof>

lemma decl_stack_top:

assumes "decl i tp a3 cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 and "Denvalue env \$\$ i = None"
 shows " $\exists t. Denvalue env' $$ i = Some (t, Stackloc (ShowL_{nat} (Toploc k)))$ "
<proof>

lemma decl_stack_topLoc:

assumes "decl i tp (Some (KStoptr p2, Storage tt)) cp cd mem sto (c, m, k, env) = Some (c', m', k', env')"
 and "Denvalue env \$\$ i = None"

```

    and "∃p. accessStore (ShowLnat (Toploc k)) k' = Some (KStoptr p)"
  shows "accessStore (ShowLnat (Toploc k)) k' = Some (KStoptr p2)"
  ⟨proof⟩

lemma decl_env_false_same_cd:
  assumes "decl i tp a3 False cd mem sto (c, m, k, env) = Some (c', m', k', env'"
  shows "c = c'"
  ⟨proof⟩

lemma decl_storage_tp_params:
  assumes "decl i (Storage tp) (Some (a3,type.Storage tp')) cp cd mem sto (c, m, k, env) = Some (c',
m', k', env'"
  shows "tp = tp'"
  ⟨proof⟩

declare decl.simps[simp del, solidity_symbex add]
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 Contracts (Contracts)

```
theory Contracts
  imports Environment
begin
```

5.1.1 Syntax of Contracts

```
datatype l = Id Identifier
           | Ref Identifier "e list"
and       e = INT bits int
           | UINT bits 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"
           | CONTRACTS

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
           | NEW Identifier "e list" e
```

5.1.2 state

```
type_synonym gas = nat

record state =
  Accounts :: accounts
  Stack    :: stack
  Memory   :: memoryT
  Storage  :: "address ⇒ storageT"
  Gas      :: gas
```

```

lemma all_gas_le:
  assumes "state.Gas x < (state.Gas y :: nat)"
    and "∀ z. state.Gas z < state.Gas y → P z → Q z"
  shows "∀ z. state.Gas z ≤ state.Gas x ∧ P z → Q z" <proof>

lemma all_gas_less:
  assumes "∀ z. state.Gas z < state.Gas y → P z"
    and "state.Gas x ≤ (state.Gas y :: nat)"
  shows "∀ z. state.Gas z < state.Gas x → P z" <proof>

definition incrementAccountContracts :: "address ⇒ state ⇒ state"
  where "incrementAccountContracts ad st = st (Accounts := (Accounts st)(ad := (Accounts st ad)(Contracts := Suc (Contracts (Accounts st ad)))))"

declare incrementAccountContracts_def [solidity_symbex]

lemma incrementAccountContracts_type[simp]:
  "Type (Accounts (incrementAccountContracts ad st) ad') = Type (Accounts st ad'"
  <proof>

lemma incrementAccountContracts_bal[simp]:
  "Bal (Accounts (incrementAccountContracts ad st) ad') = Bal (Accounts st ad'"
  <proof>

lemma incrementAccountContracts_stack[simp]:
  "Stack (incrementAccountContracts ad st) = Stack st"
  <proof>

lemma incrementAccountContracts_memory[simp]:
  "Memory (incrementAccountContracts ad st) = Memory st"
  <proof>

lemma incrementAccountContracts_storage[simp]:
  "Storage (incrementAccountContracts ad st) = Storage st"
  <proof>

lemma incrementAccountContracts_gas[simp]:
  "Gas (incrementAccountContracts ad st) = Gas st"
  <proof>

lemma gas_induct:
  assumes "∧ s. ∀ s'. Gas s' < Gas s → P s' ⇒ P s"
  shows "P s" <proof>

definition emptyStorage :: "address ⇒ storageT"
  where
    "emptyStorage _ = {$$$}"

declare emptyStorage_def [solidity_symbex]

abbreviation mystate :: state
  where "mystate ≡ (
    Accounts = emptyAccount,
    Stack = emptyStore,
    Memory = emptyTypedStore,
    Storage = emptyStorage,
    Gas = 0
  )"

datatype ex = Gas | Err

```

5.1.3 Contracts

A contract consists of methods, functions, and storage variables.

A method is a triple consisting of

- A list of formal parameters
- A flag to signal external methods
- A statement

A function is a pair consisting of

- A list of formal parameters
- A flag to signal external functions
- An expression

```
datatype (discs_sels) member = Method "(Identifier × type) list × bool × s"
      | Function "(Identifier × type) list × bool × e"
      | Var stypes
```

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

- An assignment of contract to identifiers
- A constructor
- A fallback statement which is executed after money is being transferred to the contract.

<https://docs.soliditylang.org/en/v0.8.6/Contracts.html#fallback-function>

```
type_synonym contract = "(Identifier, member) fmap × ((Identifier × type) list × s) × s"
```

```
type_synonym environmentp = "(Identifier, contract) fmap"
```

```
definition init::"(Identifier, member) fmap ⇒ Identifier ⇒ environment ⇒ environment"
  where "init ct i e = (case ct $$ i of
    Some (Var tp) ⇒ updateEnvDup i (type.Storage tp) (Storeloc i) e
    | _ ⇒ e)"
```

```
declare init_def [solidity_symbex]
```

```
lemma init_s11[simp]:
  assumes "fmlookup ct i = Some (Var tp)"
  shows "init ct i e = updateEnvDup i (type.Storage tp) (Storeloc i) e"
  <proof>
```

```
lemma init_s12[simp]:
  assumes "i |∈| fmdom (Dvalue e)"
  shows "init ct i e = e"
  <proof>
```

```
lemma init_s13[simp]:
  assumes "fmlookup ct i = Some (Var tp)"
  and "¬ i |∈| fmdom (Dvalue e)"
  shows "init ct i e = updateEnv i (type.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_s23[simp]:
  assumes "fmlookup ct i = Some (Function f)"
  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"
  <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 init_contract[simp]:
  "Contract (init ct i e) = Contract 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 ∧ Contract e' = Contract e"
  <proof>

lemma ffold_init_ad[simp]: "Address (ffold (init ct) e xs) = Address e"
  <proof>

lemma ffold_init_sender[simp]: "Sender (ffold (init ct) e xs) = Sender e"
  <proof>

lemma ffold_init_contract[simp]: "Contract (ffold (init ct) e xs) = Contract e"
  <proof>

lemma ffold_init_dom:
  "fmdom (Denvvalue (ffold (init ct) e xs)) |⊆| fmdom (Denvvalue e) |∪| xs"
  <proof>

lemma ffold_init_fmap:
  assumes "ct $$ i = Some (Var tp)"
  and "i |∉| fmdom (Denvvalue e)"
  shows "i |∈| xs ⇒ Denvvalue (ffold (init ct) e xs) $$ i = Some (type.Storage tp, Storeloc i)"
  <proof>

lemma ffold_init_fmdom:
  assumes "fmlookup ct i = Some (Var tp)"
  and "i |∉| fmdom (Denvvalue e)"
  shows "fmlookup (Denvvalue (ffold (init ct) e (fmdom ct))) i = Some (type.Storage tp, Storeloc i)"
  <proof>

lemma ffold_init_emptyDen:
  shows "fmlookup (Denvvalue (ffold (init ct) (emptyEnv a b c d) xs)) i = Some x → i |∈| xs"
  <proof>

lemma ffold_init_emptyDen_ran:
  shows "fmlookup (Denvvalue (ffold (init ct) (emptyEnv a b c d) xs)) i = Some (type.Storage tp,
Storeloc i)
  → ct $$ i = Some (Var tp) "
  <proof>

```

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

```

definition ffold_init::"(String.literal, member) fmap  $\Rightarrow$  environment  $\Rightarrow$  String.literal fset  $\Rightarrow$ 
environment" where
  <ffold_init ct a c = ffold (init ct) a c>
declare ffold_init_def [simp,solidity_symbex]

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 " $\bigwedge v. x = KValue\ v \implies f\ v\ s = f'\ v\ s$ "
  and " $\bigwedge p. x = KCDptr\ p \implies g\ p\ s = g'\ p\ s$ "
  and " $\bigwedge p. x = KMemptr\ p \implies h\ p\ s = h'\ p\ s$ "
  and " $\bigwedge p. x = KStoptr\ p \implies i\ p\ s = i'\ p\ s$ "
  shows "(case x of KValue v  $\Rightarrow$  f v | KCDptr p  $\Rightarrow$  g p | KMemptr p  $\Rightarrow$  h p | KStoptr p  $\Rightarrow$  i p) s
= (case x' of KValue v  $\Rightarrow$  f' v | KCDptr p  $\Rightarrow$  g' p | KMemptr p  $\Rightarrow$  h' p | KStoptr p  $\Rightarrow$  i' p) s"
  <proof>

lemma bind_case_type_cong [fundef_cong]:
  assumes "x = x'"
  and " $\bigwedge t. x = Value\ t \implies f\ t\ s = f'\ t\ s$ "
  and " $\bigwedge t. x = Calldata\ t \implies g\ t\ s = g'\ t\ s$ "
  and " $\bigwedge t. x = type.Memory\ t \implies h\ t\ s = h'\ t\ s$ "
  and " $\bigwedge t. x = type.Storage\ t \implies i\ t\ s = i'\ t\ s$ "
  shows "(case x of Value t  $\Rightarrow$  f t | Calldata t  $\Rightarrow$  g t | type.Memory t  $\Rightarrow$  h t | type.Storage t  $\Rightarrow$  i
t) s
= (case x' of Value t  $\Rightarrow$  f' t | Calldata t  $\Rightarrow$  g' t | type.Memory t  $\Rightarrow$  h' t | type.Storage t  $\Rightarrow$  i'
t) s"
  <proof>

lemma bind_case_denvalue_cong [fundef_cong]:
  assumes "x = x'"
  and " $\bigwedge a. x = (Stackloc\ a) \implies f\ a\ s = f'\ a\ s$ "
  and " $\bigwedge a. x = (Storeloc\ a) \implies g\ a\ s = g'\ a\ s$ "
  shows "(case x of (Stackloc a)  $\Rightarrow$  f a | (Storeloc a)  $\Rightarrow$  g a) s
= (case x' of (Stackloc a)  $\Rightarrow$  f' a | (Storeloc a)  $\Rightarrow$  g' a) s"
  <proof>

lemma bind_case_mtypes_cong [fundef_cong]:
  assumes "x = x'"
  and " $\bigwedge a\ t. x = (MArray\ a\ t) \implies f\ a\ t\ s = f'\ a\ t\ s$ "
  and " $\bigwedge p. x = (MValue\ p) \implies g\ p\ s = g'\ p\ s$ "
  shows "(case x of (MArray a t)  $\Rightarrow$  f a t | (MValue p)  $\Rightarrow$  g p) s
= (case x' of (MArray a t)  $\Rightarrow$  f' a t | (MValue p)  $\Rightarrow$  g' p) s"
  <proof>

lemma bind_case_stypes_cong [fundef_cong]:
  assumes "x = x'"
  and " $\bigwedge a\ t. x = (SArray\ a\ t) \implies f\ a\ t\ s = f'\ a\ t\ s$ "
  and " $\bigwedge a\ t. x = (SMap\ a\ t) \implies g\ a\ t\ s = g'\ a\ t\ s$ "
  and " $\bigwedge p. x = (SValue\ p) \implies h\ p\ s = h'\ p\ s$ "
  shows "(case x of (SArray a t)  $\Rightarrow$  f a t | (SMap a t)  $\Rightarrow$  g a t | (SValue p)  $\Rightarrow$  h p) s
= (case x' of (SArray a t)  $\Rightarrow$  f' a t | (SMap a t)  $\Rightarrow$  g' a t | (SValue p)  $\Rightarrow$  h' p) s"
  <proof>

lemma bind_case_types_cong [fundef_cong]:
  assumes "x = x'"
  and " $\bigwedge a. x = (TInt\ a) \implies f\ a\ s = f'\ a\ s$ "
  and " $\bigwedge a. x = (TUInt\ a) \implies g\ a\ s = g'\ a\ s$ "
  and "x = TBool  $\implies h\ s = h'\ s$ "
  and "x = TAddr  $\implies i\ s = i'\ s$ "

```

```

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 = Function a ⇒ g a s = g' a s"
  and "∧a. x = Var a ⇒ h a s = h' a s"
  shows "(case x of Method a ⇒ f a | Function a ⇒ g a | Var a ⇒ h a) s
        = (case x' of Method a ⇒ f' a | Function a ⇒ g' a | Var a ⇒ h' 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⟩

```

```

end

```

5.2 Expressions (Expressions)

```

theory Expressions

```

```

  imports Contracts StateMonad

```

```

begin

```

5.2.1 Semantics of Expressions

```

definition lift ::

```

```

  "(e ⇒ environment ⇒ calldataT ⇒ state ⇒ (stackvalue * type, ex, gas) state_monad)
  ⇒ (types ⇒ types ⇒ valuetype ⇒ valuetype ⇒ (valuetype * types) option)
  ⇒ e ⇒ e ⇒ environment ⇒ calldataT ⇒ state ⇒ (stackvalue * type, ex, gas) state_monad"

```

```

where

```

```

  "lift expr f e1 e2 e cd st ≡
  (do {
    kv1 ← expr e1 e cd st;
    (v1, t1) ← case kv1 of (KValue v1, Value t1) ⇒ return (v1, t1) | _ ⇒ (throw Err::(valuetype *
types, ex, gas) state_monad);
    kv2 ← expr e2 e cd st;
    (v2, t2) ← case kv2 of (KValue v2, Value t2) ⇒ return (v2, t2) | _ ⇒ (throw Err::(valuetype *
types, ex, gas) state_monad);
    (v, t) ← (option Err (λ_::gas. f t1 t2 v1 v2))::(valuetype * types, ex, gas) state_monad;
    return (KValue v, Value t)::(stackvalue * type, ex, gas) state_monad
  })"

```

```

declare lift_def[simp]

```

```

lemma lift_cong [fundef_cong]:

```

```

  assumes "expr e1 e cd st g = expr' e1 e cd st g"
  and "∧v g'. expr' e1 e cd st g = Normal (v,g') ⇒ expr e2 e cd st g' = expr' e2 e cd st g'"
  shows "lift expr f e1 e2 e cd st g = lift expr' f e1 e2 e cd st g"
⟨proof⟩

```

```

datatype (discs_sels) ltype

```

```

  = LStackloc location
  | LMemloc location
  | LStoreloc location

```

```

locale expressions_with_gas =

```

```

  fixes costse :: "e ⇒ environment ⇒ calldataT ⇒ state ⇒ gas"
  and ep::environmentp

```

```

    assumes call_not_zero[termination_simp]: " $\bigwedge e \text{ cd st i ix. } 0 < (\text{costs}_e (\text{CALL } i \text{ ix}) e \text{ cd st})$ "
    and ecall_not_zero[termination_simp]: " $\bigwedge e \text{ cd st a i ix. } 0 < (\text{costs}_e (\text{ECALL } a \text{ i ix}) e \text{ cd st})$ "
begin
function (domintros) msel::"bool  $\Rightarrow$  mtypes  $\Rightarrow$  location  $\Rightarrow$  e list  $\Rightarrow$  environment  $\Rightarrow$  calldataT  $\Rightarrow$  state
 $\Rightarrow$  (location * mtypes, ex, gas) state_monad"
    and ssel::"stypes  $\Rightarrow$  location  $\Rightarrow$  e list  $\Rightarrow$  environment  $\Rightarrow$  calldataT  $\Rightarrow$  state  $\Rightarrow$  (location *
stypes, ex, gas) state_monad"
    and expr::"e  $\Rightarrow$  environment  $\Rightarrow$  calldataT  $\Rightarrow$  state  $\Rightarrow$  (stackvalue * type, ex, gas) state_monad"
    and load :: "bool  $\Rightarrow$  (Identifier  $\times$  type) list  $\Rightarrow$  e list  $\Rightarrow$  environment  $\Rightarrow$  calldataT  $\Rightarrow$  stack  $\Rightarrow$ 
memoryT  $\Rightarrow$  environment  $\Rightarrow$  calldataT  $\Rightarrow$  state  $\Rightarrow$  (environment  $\times$  calldataT  $\times$  stack  $\times$  memoryT, ex, gas)
state_monad"
    and rexp::"l  $\Rightarrow$  environment  $\Rightarrow$  calldataT  $\Rightarrow$  state  $\Rightarrow$  (stackvalue * type, ex, gas) state_monad"
where
    "msel _ _ _ [] _ _ _ g = throw Err g"
| "msel _ (MTValue _) _ _ _ _ g = throw Err g"
| "msel _ (MTArray al t) loc [x] env cd st g =
    (do {
        kv  $\leftarrow$  expr x env cd st;
        (v, b)  $\leftarrow$  case kv of (KValue v, Value (TUInt b))  $\Rightarrow$  return (v, b) | _  $\Rightarrow$  throw Err;
        assert Err ( $\lambda\_.$  less (TUInt b) (TUInt b256) v (ShowLint (int al)) = Some (ShowLbool True, TBool));
        return (hash loc v, t)
    }) g"
| "msel mm (MTArray al t) loc (x # y # ys) env cd st g =
    (do {
        kv  $\leftarrow$  expr x env cd st;
        (v, b)  $\leftarrow$  case kv of (KValue v, Value (TUInt b))  $\Rightarrow$  return (v, b) | _  $\Rightarrow$  throw Err;
        assert Err ( $\lambda\_.$  less (TUInt b) (TUInt b256) v (ShowLint (int al)) = Some (ShowLbool True, TBool));
        l  $\leftarrow$  case accessStore (hash loc v) (if mm then Memory st else cd) of Some (MPointer l)  $\Rightarrow$  return l
    })  $\Rightarrow$  throw Err;
    msel mm t l (y#ys) env cd st
    }) g"
| "ssel tp loc Nil _ _ _ g = return (loc, tp) g"
| "ssel (STValue _) _ (_ # _) _ _ _ g = throw Err g"
| "ssel (STArray al t) loc (x # xs) env cd st g =
    (do {
        kv  $\leftarrow$  expr x env cd st;
        (v, b)  $\leftarrow$  case kv of (KValue v, Value (TUInt b))  $\Rightarrow$  return (v, b) | _  $\Rightarrow$  throw Err;
        assert Err ( $\lambda\_.$  less (TUInt b) (TUInt b256) v (ShowLint (int al)) = Some (ShowLbool True, TBool));
        ssel t (hash loc v) xs env cd st
    }) g"
| "ssel (STMap t t') loc (x # xs) env cd st g =
    (do {
        kv  $\leftarrow$  expr x env cd st;
        (v, t'')  $\leftarrow$  case kv of (KValue v, Value t'')  $\Rightarrow$  return (v, t'') | _  $\Rightarrow$  throw Err;
        assert Err ( $\lambda\_.$  comp t'' t);
        ssel t' (hash loc v) xs env cd st
    }) g"
| "expr (e.INT b x) e cd st g =
    (do {
        assert Gas ( $\lambda g.$  g > costse (e.INT b x) e cd st);
        modify ( $\lambda g.$  g - costse (e.INT b x) e cd st);
        return (KValue (createSInt b x), Value (TSInt b))
    }) g"
| "expr (UINT b x) e cd st g =
    (do {
        assert Gas ( $\lambda g.$  g > costse (UINT b x) e cd st);
        modify ( $\lambda g.$  g - costse (UINT b x) e cd st);
        return (KValue (createUInt b x), Value (TUInt b))
    }) g"
| "expr (ADDRESS ad) e cd st g =
    (do {
        assert Gas ( $\lambda g.$  g > costse (ADDRESS ad) e cd st);
        modify ( $\lambda g.$  g - costse (ADDRESS ad) e cd st);

```

5 Expressions and Statements

```

    return (KValue (createAddress ad), Value TAddr)
  }) g"
| "expr (BALANCE ad) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  (BALANCE ad) e cd st);
    modify ( $\lambda g. g - costs_e$  (BALANCE ad) e cd st);
    kv  $\leftarrow$  expr ad e cd st;
    adv  $\leftarrow$  case kv of (KValue adv, Value TAddr)  $\Rightarrow$  return adv | _  $\Rightarrow$  throw Err;
    return (KValue (Bal ((Accounts st) adv)), Value (TUInt b256))
  }) g"
| "expr THIS e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  THIS e cd st);
    modify ( $\lambda g. g - costs_e$  THIS e cd st);
    return (KValue (Address e), Value TAddr)
  }) g"
| "expr SENDER e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  SENDER e cd st);
    modify ( $\lambda g. g - costs_e$  SENDER e cd st);
    return (KValue (Sender e), Value TAddr)
  }) g"
| "expr VALUE e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  VALUE e cd st);
    modify ( $\lambda g. g - costs_e$  VALUE e cd st);
    return (KValue (Svalue e), Value (TUInt b256))
  }) g"
| "expr TRUE e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  TRUE e cd st);
    modify ( $\lambda g. g - costs_e$  TRUE e cd st);
    return (KValue (ShowLbool True), Value TBool)
  }) g"
| "expr FALSE e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  FALSE e cd st);
    modify ( $\lambda g. g - costs_e$  FALSE e cd st);
    return (KValue (ShowLbool False), Value TBool)
  }) g"
| "expr (NOT x) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  (NOT x) e cd st);
    modify ( $\lambda g. g - costs_e$  (NOT x) e cd st);
    kv  $\leftarrow$  expr x e cd st;
    v  $\leftarrow$  case kv of (KValue v, Value TBool)  $\Rightarrow$  return v | _  $\Rightarrow$  throw Err;
    (if v = ShowLbool True then expr FALSE e cd st
     else if v = ShowLbool False then expr TRUE e cd st
     else throw Err)
  }) g"
| "expr (PLUS e1 e2) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  (PLUS e1 e2) e cd st);
    modify ( $\lambda g. g - costs_e$  (PLUS e1 e2) e cd st);
    lift expr add e1 e2 e cd st
  }) g"
| "expr (MINUS e1 e2) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e$  (MINUS e1 e2) e cd st);
    modify ( $\lambda g. g - costs_e$  (MINUS e1 e2) e cd st);
    lift expr sub e1 e2 e cd st
  }) g"
| "expr (LESS e1 e2) e cd st g =
  (do {

```

```

    assert Gas ( $\lambda g. g > costs_e (LESS\ e1\ e2)\ e\ cd\ st$ );
    modify ( $\lambda g. g - costs_e (LESS\ e1\ e2)\ e\ cd\ st$ );
    lift expr less e1 e2 e cd st
  }) g"
| "expr (EQUAL e1 e2) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e (EQUAL\ e1\ e2)\ e\ cd\ st$ );
    modify ( $\lambda g. g - costs_e (EQUAL\ e1\ e2)\ e\ cd\ st$ );
    lift expr equal e1 e2 e cd st
  }) g"
| "expr (AND e1 e2) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e (AND\ e1\ e2)\ e\ cd\ st$ );
    modify ( $\lambda g. g - costs_e (AND\ e1\ e2)\ e\ cd\ st$ );
    lift expr vtand e1 e2 e cd st
  }) g"
| "expr (OR e1 e2) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e (OR\ e1\ e2)\ e\ cd\ st$ );
    modify ( $\lambda g. g - costs_e (OR\ e1\ e2)\ e\ cd\ st$ );
    lift expr vtor e1 e2 e cd st
  }) g"
| "expr (LVAL i) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e (LVAL\ i)\ e\ cd\ st$ );
    modify ( $\lambda g. g - costs_e (LVAL\ i)\ e\ cd\ st$ );
    rexp i e cd st
  }) g"
| "expr (CALL i xe) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e (CALL\ i\ xe)\ e\ cd\ st$ );
    modify ( $\lambda g. g - costs_e (CALL\ i\ xe)\ e\ cd\ st$ );
    (ct, _)  $\leftarrow$  option Err ( $\lambda_. ep\ \$\$ (Contract\ e)$ );
    (fp, x)  $\leftarrow$  case ct  $\$\$ i$  of Some (Function (fp, False, x))  $\Rightarrow$  return (fp, x) | _  $\Rightarrow$  throw Err;
    let e' = ffold_init ct (emptyEnv (Address e) (Contract e) (Sender e) (Svalue e)) (fmdom ct);
    (el, cdl, kl, ml)  $\leftarrow$  load False fp xe e' emptyTypedStore emptyStore (Memory st) e cd st;
    (val, typ)  $\leftarrow$  expr x el cdl (st(|Stack:=kl, Memory:=ml|));
    case val of KValue x  $\Rightarrow$  return (val, typ)
    | KCDptr cdloc  $\Rightarrow$  throw Err
    | KMemptr memloc  $\Rightarrow$  throw Err
    | KStoptr storloc  $\Rightarrow$  return (val, typ)
  }) g"
| "expr (ECALL ad i xe) e cd st g =
  (do {
    assert Gas ( $\lambda g. g > costs_e (ECALL\ ad\ i\ xe)\ e\ cd\ st$ );
    modify ( $\lambda g. g - costs_e (ECALL\ ad\ i\ xe)\ e\ cd\ st$ );
    kad  $\leftarrow$  expr ad e cd st;
    adv  $\leftarrow$  case kad of (KValue adv, Value TAddr)  $\Rightarrow$  return adv | _  $\Rightarrow$  throw Err;
    assert Err ( $\lambda_. adv \neq Address\ e$ );
    c  $\leftarrow$  case Type (Accounts st adv) of Some (atype.Contract c)  $\Rightarrow$  return c | _  $\Rightarrow$  throw Err;
    (ct, _)  $\leftarrow$  option Err ( $\lambda_. ep\ \$\$ c$ );
    (fp, x)  $\leftarrow$  case ct  $\$\$ i$  of Some (Function (fp, True, x))  $\Rightarrow$  return (fp, x) | _  $\Rightarrow$  throw Err;
    let e' = ffold_init ct (emptyEnv adv c (Address e) (ShowLnat 0)) (fmdom ct);
    (el, cdl, kl, ml)  $\leftarrow$  load True fp xe e' emptyTypedStore emptyTypedStore e cd st;
    (val, typ)  $\leftarrow$  expr x el cdl (st(|Stack:=kl, Memory:=ml|));
    case val of KValue x  $\Rightarrow$  return (val, typ)
    | KCDptr cdloc  $\Rightarrow$  throw Err
    | KMemptr memloc  $\Rightarrow$  throw Err
    | KStoptr storloc  $\Rightarrow$  throw Err
  }) g"

```

5 Expressions and Statements

```

| "load cp ((ip, tp)#pl) (ex#el) ev' cd' sck' mem' ev cd st g =
  (if (case tp of type.Storage x ⇒ cp | _ ⇒ False) then throw Err
  else
    (case Denvalue ev' $$ ip of Some x' ⇒ throw Err
    | None ⇒
    do {
      (v, t) ← expr ex ev cd st;
      (c, m, k, e) ← (case decl ip tp (Some (v,t)) cp cd (Memory st) ((Storage st) (Address ev))
      (cd', mem', sck', ev')
        of Some (c, m, k, e) ⇒ return (c, m, k, e)
        | None ⇒ throw Err);
      load cp pl el e c k m ev cd st}}) g"
| "load _ [] (#_) _ _ _ _ _ _ _ _ g = throw Err g"
| "load _ (#_) [] _ _ _ _ _ _ _ _ g = throw Err g"
| "load _ [] [] ev' cd' sck' mem' ev cd st g = return (ev', cd', sck', mem') g"

| "rexp (Id i) e cd st g =
  (case fmlookup (Denvalue e) i of
  Some (tp, Stackloc l) ⇒
    (case accessStore l (Stack st) 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 (type.Storage (STValue t), Storeloc l) ⇒ return (KValue (accessStorage t l (Storage st
  (Address e))), Value t)
  | Some (type.Storage (STArray x t), Storeloc l) ⇒ return (KStoptr l, type.Storage (STArray x t))
  | _ ⇒ throw Err) g"
| "rexp (Ref i r) e cd st g =
  (case fmlookup (Denvalue e) i of
  Some (tp, (Stackloc l)) ⇒
    (case accessStore l (Stack st) of
    Some (KCDptr l') ⇒
      do {
        t ← case tp of Calldata t ⇒ return t | _ ⇒ throw Err;
        (l'', t') ← msel False t l' r e cd st;
        (case t' of
        MTValue t'' ⇒
          do {
            v ← case accessStore l'' cd of Some (MValue v) ⇒ return v | _ ⇒ throw Err;
            return (KValue v, Value t'')
          }
        | MTArray x t'' ⇒
          do {
            p ← case accessStore l'' cd of Some (MPointer p) ⇒ return p | _ ⇒ throw Err;
            return (KCDptr p, Calldata (MTArray x t''))
          }
        )
      }
  | Some (KMemptr l') ⇒
    do {
      t ← case tp of type.Memory t ⇒ return t | _ ⇒ throw Err;
      (l'', t') ← msel True t l' r e cd st;
      (case t' of
      MTValue t'' ⇒
        do {
          v ← case accessStore l'' (Memory st) of Some (MValue v) ⇒ return v | _ ⇒ throw
Err;
          return (KValue v, Value t'')
        }
      | MTArray x t'' ⇒
        do {
          p ← case accessStore l'' (Memory st) of Some (MPointer p) ⇒ return p | _ ⇒ throw

```

```

Err;
      return (KMemptr p, type.Memory (MArray x t'))
    }
  )
}
| Some (KStoptr l') ⇒
do {
  t ← case tp of type.Storage t ⇒ return t | _ ⇒ throw Err;
  (l'', t') ← ssel t l' r e cd st;
  (case t' of
    STValue t'' ⇒ return (KValue (accessStorage t'' l'' (Storage st (Address e))), Value
t'')
    | STArray _ _ ⇒ return (KStoptr l'', type.Storage t')
    | STMap _ _ ⇒ return (KStoptr l'', type.Storage t'))
  }
| _ ⇒ throw Err)
| Some (tp, (Storeloc l)) ⇒
do {
  t ← case tp of type.Storage t ⇒ return t | _ ⇒ throw Err;
  (l', t') ← ssel t l r e cd st;
  (case t' of
    STValue t'' ⇒ return (KValue (accessStorage t'' l' (Storage st (Address e))), Value t'')
    | STArray _ _ ⇒ return (KStoptr l', type.Storage t')
    | STMap _ _ ⇒ return (KStoptr l', type.Storage t'))
  }
| None ⇒ throw Err) g"
| "expr CONTRACTS e cd st g =
(do {
  assert Gas (λg. g > costse CONTRACTS e cd st);
  modify (λg. g - costse CONTRACTS e cd st);
  prev ← case Contracts (Accounts st (Address e)) of 0 ⇒ throw Err | Suc n ⇒ return n;
  return (KValue (hashversion (Address e) (ShowLnat prev)), Value TAddr)
}) g"
⟨proof⟩

```

5.2.2 Termination

To prove termination we first need to show that expressions do not increase gas

lemma lift_gas:

```

assumes "lift expr f e1 e2 e cd st g = Normal (v, g'"
and "∧v g'. expr e1 e cd st g = Normal (v, g') ⇒ g' ≤ g"
and "∧v g' v' t' g''. expr e1 e cd st g = Normal (v, g')
⇒ expr e2 e cd st g' = Normal (v', g'')
⇒ g'' ≤ g'"
shows "g' ≤ g"

```

⟨proof⟩

lemma msel_ssel_expr_load_rexp_dom_gas[rule_format]:

```

"msel_ssel_expr_load_rexp_dom (Inl (Inl (c1, t1, l1, xe1, ev1, cd1, st1, g1)))
⇒ (∀v1' g1'. msel c1 t1 l1 xe1 ev1 cd1 st1 g1 = Normal (v1', g1') → g1' ≤ g1)"
"msel_ssel_expr_load_rexp_dom (Inl (Inr (t2, l2, xe2, ev2, cd2, st2, g2)))
⇒ (∀v2' g2'. ssel t2 l2 xe2 ev2 cd2 st2 g2 = Normal (v2', g2') → g2' ≤ g2)"
"msel_ssel_expr_load_rexp_dom (Inr (Inl (e4, ev4, cd4, st4, g4)))
⇒ (∀v4' g4'. expr e4 ev4 cd4 st4 g4 = Normal (v4', g4') → g4' ≤ g4)"
"msel_ssel_expr_load_rexp_dom (Inr (Inr (Inl (lcp, lis, lxs, lev0, lcd0, lk, lm, lev, lcd, lst,
lg))))
⇒ (∀ev cd k m g'. load lcp lis lxs lev0 lcd0 lk lm lev lcd lst lg = Normal ((ev, cd, k, m),
g') → g' ≤ lg ∧ Address ev = Address lev0 ∧ Sender ev = Sender lev0 ∧ Svalue ev = Svalue lev0 ∧
Contract ev = Contract lev0)"
"msel_ssel_expr_load_rexp_dom (Inr (Inr (Inr (l3, ev3, cd3, st3, g3))))
⇒ (∀v3' g3'. rexp l3 ev3 cd3 st3 g3 = Normal (v3', g3') → g3' ≤ g3)"

```

⟨proof⟩

Now we can define the termination function

```

fun mgas
  where "mgas (Inr (Inr (Inr l))) = snd (snd (snd (snd l)))"
        | "mgas (Inr (Inr (Inl l))) = snd (snd (snd (snd (snd (snd (snd (snd (snd (snd l)))))))))"
        | "mgas (Inr (Inl l)) = snd (snd (snd l))"
        | "mgas (Inl (Inr l)) = snd (snd (snd (snd (snd l))))"
        | "mgas (Inl (Inl l)) = snd (snd (snd (snd (snd (snd l)))))"

fun msize
  where "msize (Inr (Inr (Inr l))) = size (fst l)"
        | "msize (Inr (Inr (Inl l))) = size_list size (fst (snd (snd l)))"
        | "msize (Inr (Inl l)) = size (fst l)"
        | "msize (Inl (Inr l)) = size_list size (fst (snd (snd l)))"
        | "msize (Inl (Inl l)) = size_list size (fst (snd (snd (snd l))))"

method msel_ssel_expr_load_rexp_dom =
  match premises in e: "expr _ _ _ _ = Normal (_,_)" and d[thin]: "msel_ssel_expr_load_rexp_dom (Inr (Inl _))" ⇒ <insert msel_ssel_expr_load_rexp_dom_gas(3)[OF d e]> |
  match premises in l: "load _ _ _ _ _ _ = Normal (_,_)" and d[thin]:
  "msel_ssel_expr_load_rexp_dom (Inr (Inr (Inl _)))" ⇒ <insert msel_ssel_expr_load_rexp_dom_gas(4)[OF d l, THEN conjunct1]>

method costs =
  match premises in "costs_e (CALL i xe) e cd st <_" for i xe and e::environment and cd::calldataT and st::state ⇒ <insert call_not_zero[of (unchecked) i xe e cd st]> |
  match premises in "costs_e (ECALL ad i xe) e cd st <_" for ad i xe and e::environment and cd::calldataT and st::state ⇒ <insert ecall_not_zero[of (unchecked) ad i xe e cd st]>

termination msel
  <proof>

```

5.2.3 gas Reduction

The following corollary is a generalization of `msel_ssel_expr_load_rexp_dom_gas`. We first prove that the function is defined for all input values and then obtain the final result as a corollary.

```

lemma msel_ssel_expr_load_rexp_dom:
  "msel_ssel_expr_load_rexp_dom (Inl (Inl (c1, t1, l1, xe1, ev1, cd1, st1, g1)))"
  "msel_ssel_expr_load_rexp_dom (Inl (Inr (t2, l2, xe2, ev2, cd2, st2, g2)))"
  "msel_ssel_expr_load_rexp_dom (Inr (Inl (e4, ev4, cd4, st4, g4)))"
  "msel_ssel_expr_load_rexp_dom (Inr (Inr (Inl (lcp, lis, lxs, lev0, lcd0, lk, lm, lev, lcd, lst, lg))))"
  "msel_ssel_expr_load_rexp_dom (Inr (Inr (Inr (l3, ev3, cd3, st3, g3))))"
  <proof>

```

```

lemmas msel_ssel_expr_load_rexp_gas =
  msel_ssel_expr_load_rexp_dom_gas(1)[OF msel_ssel_expr_load_rexp_dom(1)]
  msel_ssel_expr_load_rexp_dom_gas(2)[OF msel_ssel_expr_load_rexp_dom(2)]
  msel_ssel_expr_load_rexp_dom_gas(3)[OF msel_ssel_expr_load_rexp_dom(3)]
  msel_ssel_expr_load_rexp_dom_gas(4)[OF msel_ssel_expr_load_rexp_dom(4)]
  msel_ssel_expr_load_rexp_dom_gas(5)[OF msel_ssel_expr_load_rexp_dom(5)]

```

```

lemma expr_sender:
  assumes "expr SENDER e cd st g = Normal ((KValue adv, Value TAddr), g)"
  shows "adv = Sender e" <proof>

```

```

declare expr.simps[simp del, solidity_symbex add]
declare load.simps[simp del, solidity_symbex add]
declare ssel.simps[simp del, solidity_symbex add]
declare msel.simps[simp del, solidity_symbex add]
declare rexp.simps[simp del, solidity_symbex add]

```

```

lemma load_denvai_sub[rule_format]:

```

```

"∀ l1' t1' g1' arr. msel c1 t1 l1 xe1 ev1 cd1 st1 g1 = Normal ((l1', t1'), g1') → True"
"∀ l2' v2' t2' g2'. ssel t2 l2 xe2 ev2 cd2 st2 g2 = Normal ((l2', t2'), g2') → True"
"∀ v t g4'. expr e4 ev4 cd4 st4 g4 = Normal ((v, t), g4') → True"

"∀ ev cd k m g'. load lcp lis lxs lev0 lcd0 lk lm lev lcd lst lg = Normal ((ev, cd, k, m), g') →
fmdom (Denvalue lev0) |⊆| fmdom (Denvalue ev) "
"∀ v3' t3' g3'. rexp l3 ev3 cd3 st3 g3 = Normal ((v3', t3'), g3') → True"
⟨proof⟩

lemma load_denval_existing_remain[rule_format]:
"∀ l1' t1' g1' arr. msel c1 t1 l1 xe1 ev1 cd1 st1 g1 = Normal ((l1', t1'), g1') → True"
"∀ l2' v2' t2' g2'. ssel t2 l2 xe2 ev2 cd2 st2 g2 = Normal ((l2', t2'), g2') → True"
"∀ v t g4'. expr e4 ev4 cd4 st4 g4 = Normal ((v, t), g4') → True"

"∀ ev cd k m g' id x. load lcp lis lxs lev0 lcd0 lk lm lev lcd lst lg = Normal ((ev, cd, k, m), g') ∧
Denvalue lev0 $$ id = Some x → (Denvalue ev) $$ id = Some x "
"∀ v3' t3' g3'. rexp l3 ev3 cd3 st3 g3 = Normal ((v3', t3'), g3') → True"
⟨proof⟩

end

end

```

5.3 Statements (Statements)

theory Statements

imports Expressions StateMonad

begin

```

locale statement_with_gas = expressions_with_gas +
  fixes costs :: "s ⇒ environment ⇒ calldataT ⇒ state ⇒ gas"
  assumes while_not_zero[termination_simp]: "∧ e cd st ex s0. 0 < (costs (WHILE ex s0) e cd st) "
  and invoke_not_zero[termination_simp]: "∧ e cd st i xe. 0 < (costs (INVOKE i xe) e cd st)"
  and external_not_zero[termination_simp]: "∧ e cd st ad i xe val. 0 < (costs (EXTERNAL ad i xe
val) e cd st)"
  and transfer_not_zero[termination_simp]: "∧ e cd st ex ad. 0 < (costs (TRANSFER ad ex) e cd st)"
  and new_not_zero[termination_simp]: "∧ e cd st i xe val. 0 < (costs (NEW i xe val) e cd st)"
begin

```

5.3.1 Semantics of left expressions

We first introduce lexp.

```

fun lexp :: "l ⇒ environment ⇒ calldataT ⇒ state ⇒ (ltype * type, ex, gas) state_monad"
  where "lexp (Id i) e _ st g =
  (case (Denvalue e) $$ i of
    Some (tp, (Stackloc l)) ⇒ return (LStackloc l, tp)
  | Some (tp, (Storeloc l)) ⇒ return (LStoreloc l, tp)
  | _ ⇒ throw Err) g"
| "lexp (Ref i r) e cd st g =
  (case (Denvalue e) $$ i of
    Some (tp, Stackloc l) ⇒
      (case accessStore l (Stack st) of
        Some (KCDptr _) ⇒ throw Err
      | Some (KMemptr l') ⇒
        do {
          t ← (case tp of type.Memory t ⇒ return t | _ ⇒ throw Err);
          (l'', t') ← msel True t l' r e cd st;
          return (LMemloc l'', type.Memory t')
        }
      )
  | Some (KStoptr l') ⇒
    do {
      t ← (case tp of type.Storage t ⇒ return t | _ ⇒ throw Err);
      (l'', t') ← ssel t l' r e cd st;

```

```

    return (LStoreloc l'', type.Storage t')
  }
  | Some (KValue _) ⇒ throw Err
  | None ⇒ throw Err
| Some (tp, Storeloc l) ⇒
  do {
    t ← (case tp of type.Storage t ⇒ return t | _ ⇒ throw Err);
    (l', t') ← ssel t l r e cd st;
    return (LStoreloc l', type.Storage t')
  }
| None ⇒ throw Err) g"

```

lemma `lexp_gas[rule_format]`:

" $\forall l5' t5' g5'. \text{lexp } l5 \text{ ev5 } cd5 \text{ st5 } g5 = \text{Normal } ((l5', t5'), g5') \longrightarrow g5' \leq g5$ "
<proof>

5.3.2 Semantics of statements

The following is a helper function to connect the gas monad with the state monad.

fun

```

toState :: "(state ⇒ ('a, 'e, gas) state_monad) ⇒ ('a, 'e, state) state_monad" where
"toState gm = (λs. case gm s (state.Gas s) of
  Normal (a,g) ⇒ Normal(a,s(|Gas:=g|))
  | Exception e ⇒ Exception e)"

```

lemma `wptoState[wprule]`:

```

assumes "∧a g. gm s (state.Gas s) = Normal (a, g) ⇒ P a (s(|Gas:=g|))"
and "∧e. gm s (state.Gas s) = Exception e ⇒ E e"
shows "wp (toState gm) P E s"
<proof>

```

Now we define the semantics of statements.

function `(domintros) stmt :: "s ⇒ environment ⇒ calldataT ⇒ (unit, ex, state) state_monad"`

```

where "stmt SKIP e cd st =
  (do {
    assert Gas (λst. state.Gas st > costs SKIP e cd st);
    modify (λst. st(|Gas := state.Gas st - costs SKIP e cd st|))
  }) st"
| "stmt (ASSIGN lv ex) env cd st =
  (do {
    assert Gas (λst. state.Gas st > costs (ASSIGN lv ex) env cd st);
    modify (λst. st(|Gas := state.Gas st - costs (ASSIGN lv ex) env cd st|));
    re ← toState (expr ex env cd);
    case re of
      (KValue v, Value t) ⇒
        do {
          r1 ← toState (lexp lv env cd);
          case r1 of
            (LStackloc l, Value t') ⇒
              do {
                v' ← option Err (λ_. convert t t' v);
                modify (λst. st(|Stack := updateStore l (KValue v') (Stack st)|))
              }
            | (LStoreloc l, type.Storage (STValue t')) ⇒
              do {
                v' ← option Err (λ_. convert t t' v);
                modify (λst. st(|Storage := (Storage st) (Address env := fmupd l v' (Storage st
(Address env))))|))
              }
            | (LMemloc l, type.Memory (MTValue t')) ⇒
              do {
                v' ← option Err (λ_. convert t t' v);
                modify (λst. st(|Memory := updateStore l (MValue v') (Memory st)|))
              }
          }

```

```

    }
  | _ => throw Err
}
| (KCDptr p, Calldata (MArray x t)) =>
do {
  rl ← toState (lexp lv env cd);
  case rl of
  (LStackloc l, type.Memory t') =>
    (if t' = (MArray x t) then
      do {
        sv ← applyf (λst. accessStore l (Stack st));
        p' ← case sv of Some (KMemptr p') => return p' | _ => throw Err;
        sv' ← applyf (λst. updateStore l (KMemptr (ShowLnat (Toploc (Memory st)))) (Stack
st));
        m ← option Err (λst. cpm2m p (ShowLnat (Toploc (Memory st))) x t cd (snd
(allocate(Memory st))));
        modify (λst. st (|Memory := m, Stack := sv'))
      }
    else throw Err)
  | (LStackloc l, type.Storage t') =>
    (if cps2mTypeCompatible t' (MArray x t) then
      do {
        sv ← applyf (λst. accessStore l (Stack st));
        p' ← case sv of Some (KStoptr p') => return p' | _ => throw Err;
        s ← option Err (λst. cpm2s p p' x t cd (Storage st (Address env)));
        modify (λst. st (|Storage := (Storage st) (Address env := s)))
      } else throw Err)
  | (LStoreloc l, type.Storage t') =>
    (if cps2mTypeCompatible t' (MArray x t) then
      do {
        s ← option Err (λst. cpm2s p l x t cd (Storage st (Address env)));
        modify (λst. st (|Storage := (Storage st) (Address env := s)))
      } else throw Err)
  | (LMemloc l, type.Memory t') =>
    (if t' = (MArray x t) then
      do {
        m ← option Err (λst. cpm2m p (ShowLnat (Toploc (Memory st))) x t cd (snd
(allocate(Memory st))));
        m' ← applyf (λst. updateStore l (MPointer ((ShowLnat (Toploc (Memory st)))) m);
        modify (λst. st (|Memory := m'))
      }
    else throw Err)
  | _ => throw Err
}
| (KMemptr p, type.Memory (MArray x t)) =>
do {
  rl ← toState (lexp lv env cd);
  case rl of
  (LStackloc l, type.Memory t') =>
    (if t' = (MArray x t) then
      modify (λst. st (|Stack := updateStore l (KMemptr p) (Stack st)))
    else throw Err)
  | (LStackloc l, type.Storage t') =>
    (if cps2mTypeCompatible t' (MArray x t) then
      do {
        sv ← applyf (λst. accessStore l (Stack st));
        p' ← case sv of Some (KStoptr p') => return p' | _ => throw Err;
        s ← option Err (λst. cpm2s p p' x t (Memory st) (Storage st (Address env)));
        modify (λst. st (|Storage := (Storage st) (Address env := s)))
      } else throw Err)
  | (LStoreloc l, type.Storage t') =>
    (if cps2mTypeCompatible t' (MArray x t) then
      do {

```

```

        s ← option Err (λst. cpm2s p l x t (Memory st) (Storage st (Address env)));
        modify (λst. st (Storage := (Storage st) (Address env := s)))
      } else throw Err)
    | (LMemloc l, type.Memory t') ⇒
      (if t' = (MArray x t) then
        modify (λst. st (Memory := updateStore l (MPointer p) (Memory st)))
      else throw Err)
    | _ ⇒ throw Err
  }
| (KStoptr p, type.Storage (STArray x t)) ⇒
  do {
    r1 ← toState (lexp lv env cd);
    case r1 of
      (LStackloc l, type.Memory t') ⇒
        (if cps2mTypeCompatible (STArray x t) t' then
          do {
            sv ← applyf (λst. accessStore l (Stack st));
            p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
            sv' ← applyf (λst. updateStore l (KMemptr (ShowLnat (Toploc (Memory st)))) (Stack
st));
            m ← option Err (λst. cps2m p (ShowLnat (Toploc (Memory st))) x t (Storage st
(Address env) (snd (allocate (Memory st))));
            modify (λst. st (Memory := m, Stack := sv'))
          } else throw Err)
        | (LStackloc l, type.Storage t') ⇒
          (if t' = (STArray x t) then
            modify (λst. st (Stack := updateStore l (KStoptr p) (Stack st)))
          else throw Err)
        | (LStoreloc l, type.Storage t') ⇒
          (if t' = STArray x t then
            do {
              s ← option Err (λst. copy p l x t (Storage st (Address env)));
              modify (λst. st (Storage := (Storage st) (Address env := s)))
            } else throw Err)
          | (LMemloc l, type.Memory t') ⇒
            (if cps2mTypeCompatible (STArray x t) t' then
              do {
                m ← option Err (λst. cps2m p (ShowLnat (Toploc (Memory st))) x t (Storage st
(Address env) (snd (allocate (Memory st))));
                m' ← applyf (λst. updateStore l (MPointer ((ShowLnat (Toploc (Memory st)))) m);
                modify (λst. st (Memory := m'))
              } else throw Err)
            | _ ⇒ throw Err
          }
  }
| (KStoptr p, type.Storage (STMap t t')) ⇒
  do {
    r1 ← toState (lexp lv env cd);
    (l, t'') ← case r1 of (LStackloc l, type.Storage t'') ⇒ return (l, t'') | _ ⇒ throw Err;
    (if t'' = STMap t t' then
      modify (λst. st (Stack := updateStore l (KStoptr p) (Stack st)))
    else throw Err)
  }
| _ ⇒ throw Err
}) st"
| "stmt (COMP s1 s2) e cd st =
  (do {
    assert Gas (λst. state.Gas st > costs (COMP s1 s2) e cd st);
    modify (λst. st (state.Gas := state.Gas st - costs (COMP s1 s2) e cd st));
    stmt s1 e cd;
    stmt s2 e cd
  }) st"
| "stmt (ITE ex s1 s2) e cd st =
  (do {
    assert Gas (λst. state.Gas st > costs (ITE ex s1 s2) e cd st);

```

```

modify (λst. st(|state.Gas := state.Gas st - costs (ITE ex s1 s2) e cd st));
v ← toState (expr ex e cd);
b ← (case v of (KValue b, Value TBool) ⇒ return b | _ ⇒ throw Err);
if b = ShowLbool True then stmt s1 e cd
else if b = ShowLbool False then stmt s2 e cd
else throw Err
}) st"
| "stmt (WHILE ex s0) e cd st =
(do {
  assert Gas (λst. state.Gas st > costs (WHILE ex s0) e cd st);
  modify (λst. st(|state.Gas := state.Gas st - costs (WHILE ex s0) e cd st));
  v ← toState (expr ex e cd);
  b ← (case v of (KValue b, Value TBool) ⇒ return b | _ ⇒ throw Err);
  if b = ShowLbool True then
    do {
      stmt s0 e cd;
      stmt (WHILE ex s0) e cd
    }
  else if b = ShowLbool False then return ()
  else throw Err
}) st"
| "stmt (INVOKE i xe) e cd st =
(do {
  assert Gas (λst. state.Gas st > costs (INVOKE i xe) e cd st);
  modify (λst. st(|state.Gas := state.Gas st - costs (INVOKE i xe) e cd st));
  (ct, _) ← option Err (λ_. ep $$ Contract e);
  (fp, f) ← case ct $$ i of Some (Method (fp, False, f)) ⇒ return (fp, f) | _ ⇒ throw Err;
  let e' = ffold_init ct (emptyEnv (Address e) (Contract e) (Sender e) (Svalue e)) (fmdom ct);
  mo ← applyf Memory;
  (el, cdl, kl, ml) ← toState (load False fp xe e' emptyTypedStore emptyStore mo e cd);
  ko ← applyf Stack;
  modify (λst. st(|Stack:=kl, Memory:=ml));
  stmt f el cdl;
  modify (λst. st(|Stack:=ko))
}) st"

| "stmt (EXTERNAL ad i xe val) e cd st =
(do {
  assert Gas (λst. state.Gas st > costs (EXTERNAL ad i xe val) e cd st);
  modify (λst. st(|state.Gas := state.Gas st - costs (EXTERNAL ad i xe val) e cd st));
  kad ← toState (expr ad e cd);
  adv ← case kad of (KValue adv, Value TAddr) ⇒ return adv | _ ⇒ throw Err;
  assert Err (λ_. adv ≠ Address e);
  c ← (λst. case Type (Accounts st adv) of Some (atype.Contract c) ⇒ return c st | _ ⇒ throw Err
st);
  (ct, _, fb) ← option Err (λ_. ep $$ c);
  kv ← toState (expr val e cd);
  (v, t) ← case kv of (KValue v, Value t) ⇒ return (v, t) | _ ⇒ throw Err;
  v' ← option Err (λ_. convert t (TUInt b256) v);
  let e' = ffold_init ct (emptyEnv adv c (Address e) v') (fmdom ct);
  case ct $$ i of
    Some (Method (fp, True, f)) ⇒
      do {
        (el, cdl, kl, ml) ← toState (load True fp xe e' emptyTypedStore emptyStore emptyTypedStore
e cd);
        acc ← option Err (λst. transfer (Address e) adv v' (Accounts st));
        (ko, mo) ← applyf (λst. (Stack st, Memory st));
        modify (λst. st(|Accounts := acc, Stack:=kl, Memory:=ml));
        stmt f el cdl;
        modify (λst. st(|Stack:=ko, Memory := mo))
      }
  | None ⇒

```

```

do {
  acc ← option Err (λst. transfer (Address e) adv v' (Accounts st));
  (ko, mo) ← applyf (λst. (Stack st, Memory st));
  modify (λst. st(|Accounts := acc, Stack:=emptyStore, Memory:=emptyTypedStore));
  stmt fb e' emptyTypedStore;
  modify (λst. st(|Stack:=ko, Memory := mo))
}
| _ ⇒ throw Err
}) st"
| "stmt (TRANSFER ad ex) e cd st =
  (do {
    assert Gas (λst. state.Gas st > costs (TRANSFER ad ex) e cd st);
    modify (λst. st(|state.Gas := state.Gas st - costs (TRANSFER ad ex) e cd st));
    kv ← toState (expr ad e cd);
    adv ← case kv of (KValue adv, Value TAddr) ⇒ return adv | _ ⇒ throw Err;
    kv' ← toState (expr ex e cd);
    (v, t) ← case kv' of (KValue v, Value t) ⇒ return (v, t) | _ ⇒ throw Err;
    v' ← option Err (λ_. convert t (TUInt b256) v);
    acc ← applyf Accounts;
    case Type (acc adv) of
      Some (atype.Contract c) ⇒
        do {
          (ct, _, f) ← option Err (λ_. ep $$ c);
          let e' = ffold_init ct (emptyEnv adv c (Address e) v') (fdom ct);
          (ko, mo) ← applyf (λst. (Stack st, Memory st));
          acc' ← option Err (λst. transfer (Address e) adv v' (Accounts st));
          modify (λst. st(|Accounts := acc', Stack:=emptyStore, Memory:=emptyTypedStore));
          stmt f e' emptyTypedStore;
          modify (λst. st(|Stack:=ko, Memory := mo))
        }
      | Some EOA ⇒
        do {
          acc' ← option Err (λst. transfer (Address e) adv v' (Accounts st));
          modify (λst. st(|Accounts := acc'))
        }
      | None ⇒ throw Err
    }) st"
| "stmt (BLOCK (id0, tp, None) s) ev cd st =
  (do {
    assert Gas (λst. state.Gas st > costs (BLOCK (id0, tp, None) s) ev cd st);
    modify (λst. st(|state.Gas := state.Gas st - costs (BLOCK (id0, tp, None) s) ev cd st));
    (cd', mem', sck', e') ← option Err (λst. decl id0 tp None False cd (Memory st) (Storage st
(Address ev)) (cd, Memory st, Stack st, ev));
    modify (λst. st(|Stack := sck', Memory := mem'));
    stmt s e' cd'
  }) st"
| "stmt (BLOCK (id0, tp, Some ex') s) ev cd st =
  (do {
    assert Gas (λst. state.Gas st > costs (BLOCK (id0, tp, Some ex') s) ev cd st);
    modify (λst. st(|state.Gas := state.Gas st - costs (BLOCK (id0, tp, Some ex') s) ev cd st));
    (v, t) ← toState (expr ex' ev cd);
    (cd', mem', sck', e') ← option Err (λst. decl id0 tp (Some (v, t)) False cd (Memory st) (Storage
st (Address ev)) (cd, Memory st, Stack st, ev));
    modify (λst. st(|Stack := sck', Memory := mem'));
    stmt s e' cd'
  }) st"

| "stmt (NEW i xe val) e cd st =
  (do {
    assert Gas (λst. state.Gas st > costs (NEW i xe val) e cd st);
    modify (λst. st(|state.Gas := state.Gas st - costs (NEW i xe val) e cd st));
    adv ← applyf (λst. hash_version (Address e) (ShowLnat (Contracts (Accounts st (Address e)))));
    assert Err (λst. Type (Accounts st adv) = None);
    kv ← toState (expr val e cd);

```

```

(v, t) ← case kv of (KValue v, Value t) ⇒ return (v, t) | _ ⇒ throw Err;
v' ← option Err (λ_. convert t (TUInt b256) v);
(ct, cn, _) ← option Err (λ_. ep $$ i);
modify (λst. st(|Accounts := (Accounts st)(adv := (|Bal = ShowLint 0, Type = Some (atype.Contract
i), Contracts = 0)), Storage:=(Storage st)(adv := {$$})),
let e' = ffold_init ct (emptyEnv adv i (Address e) v') (fmdom ct);
(ei, cdi, ki, mi) ← toState (load True (fst cn) xe e' emptyTypedStore emptyStore emptyTypedStore
e cd);
acc ← option Err (λst. transfer (Address e) adv v' (Accounts st));
(ko, mo) ← applyf (λst. (Stack st, Memory st));
modify (λst. st(|Accounts := acc, Stack:=ki, Memory:=mi));
stmt (snd cn) ei cdi;
modify (λst. st(|Stack:=ko, Memory := mo));
modify (incrementAccountContracts (Address e))
}) st"
<proof>

```

5.3.3 Termination

Again, to prove termination we need a lemma regarding gas consumption.

```

lemma stmt_dom_gas[rule_format]:
  "stmt_dom (s6, ev6, cd6, st6) ⇒ (∀st6'. stmt s6 ev6 cd6 st6 = Normal((), st6') → state.Gas st6'
≤ state.Gas st6)"
<proof>

```

5.3.4 Termination function

Now we can prove termination using the lemma above.

```

fun sgas
  where "sgas l = state.Gas (snd (snd (snd l)))"

fun ssize
  where "ssize l = size (fst l)"

method stmt_dom_gas =
  match premises in s: "stmt _ _ _ = Normal (_,_)" and d[thin]: "stmt_dom _" ⇒ <insert
stmt_dom_gas[OF d s]>
method msel_ssel_expr_load_rexp =
  match premises in e[thin]: "expr _ _ _ _ = Normal (_,_)" ⇒ <insert msel_ssel_expr_load_rexp_gas(3)[OF
e]> |
  match premises in l[thin]: "load _ _ _ _ _ _ _ _ = Normal (_,_)" ⇒ <insert
msel_ssel_expr_load_rexp_gas(4)[OF l, THEN conjunct1]>
method costs =
  match premises in "costs (WHILE ex s0) e cd st <_" for ex s0 and e::environment and cd::calldataT
and st::state ⇒ <insert while_not_zero[of (unchecked) ex s0 e cd st]> |
  match premises in "costs (INVOKE i xe) e cd st <_" for i xe and e::environment and cd::calldataT
and st::state ⇒ <insert invoke_not_zero[of (unchecked) i xe e cd st]> |
  match premises in "costs (EXTERNAL ad i xe val) e cd st <_" for ad i xe val and e::environment and
cd::calldataT and st::state ⇒ <insert external_not_zero[of (unchecked) ad i xe val e cd st]> |
  match premises in "costs (TRANSFER ad ex) e cd st <_" for ad ex and e::environment and
cd::calldataT and st::state ⇒ <insert transfer_not_zero[of (unchecked) ad ex e cd st]> |
  match premises in "costs (NEW i xe val) e cd st <_" for i xe val and e::environment and
cd::calldataT and st::state ⇒ <insert new_not_zero[of (unchecked) i xe val e cd st]>

termination stmt
  <proof>

```

5.3.5 Gas Reduction

The following corollary is a generalization of `msel_ssel_expr_load_rexp_dom_gas`. We first prove that the function is defined for all input values and then obtain the final result as a corollary.

lemma stmt_dom: "stmt_dom (s6, ev6, cd6, st6)"

<proof>

lemmas stmt_gas = stmt_dom_gas[OF stmt_dom]

lemma skip:

assumes "stmt SKIP ev cd st = Normal (x, st)"

shows "state.Gas st > costs SKIP ev cd st"

and "st' = st(|state.Gas := state.Gas st - costs SKIP ev cd st|)"

<proof>

lemma assign:

assumes "stmt (ASSIGN lv ex) ev cd st = Normal (xx, st)"

obtains (1) v t g l t' g' v'

where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st|)) (state.Gas st - costs (ASSIGN lv ex) ev cd st) = Normal ((KValue v, Value t), g)"

and "lexp lv ev cd (st(|state.Gas := g|)) g = Normal((LStackloc l, Value t'),g)"

and "convert t t' v = Some v'"

and "st' = st(|state.Gas := g', Stack := updateStore l (KValue v') (Stack st)|)"

| (2) v t g l t' g' v'

where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st|)) (state.Gas st - costs (ASSIGN lv ex) ev cd st) = Normal ((KValue v, Value t), g)"

and "lexp lv ev cd (st(|state.Gas := g|)) g = Normal((LStoreloc l, type.Storage (STValue t')),g)"

and "convert t t' v = Some v'"

and "st' = st(|state.Gas := g', Storage := (Storage st) (Address ev := (fmupd l v' (Storage st (Address ev))))|)"

| (3) v t g l t' g' v'

where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st|)) (state.Gas st - costs (ASSIGN lv ex) ev cd st) = Normal ((KValue v, Value t), g)"

and "lexp lv ev cd (st(|state.Gas := g|)) g = Normal((LMemloc l, type.Memory (MTValue t')),g)"

and "convert t t' v = Some v'"

and "st' = st(|state.Gas := g', Memory := updateStore l (MValue v') (Memory st)|)"

| (4) p x t g l g' p' m t' st''

where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st|)) (state.Gas st - costs (ASSIGN lv ex) ev cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"

and "lexp lv ev cd (st(|state.Gas := g|)) g = Normal((LStackloc l, type.Memory t'),g)"

and "(MTArray x t) = t'"

and "accessStore l (Stack st) = Some (KMemptr p)"

and "(updateStore l (KMemptr (ShowL_{nat} (Toploc (Memory st)))) (Stack st)) = st''"

and "cpm2m p (ShowL_{nat} (Toploc (Memory st))) x t cd (snd (allocate (Memory st))) = Some m"

and "st' = st(|state.Gas := g', Memory := m, Stack := st''|)"

| (5) p x t g l t' g' p' s

where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st|)) (state.Gas st - costs (ASSIGN lv ex) ev cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"

and "lexp lv ev cd (st(|state.Gas := g|)) g = Normal((LStackloc l, type.Storage t'),g)"

and "accessStore l (Stack st) = Some (KStoptr p)"

and "cps2mTypeCompatible t' (MTArray x t)"

and "cpm2s p p' x t cd (Storage st (Address ev)) = Some s"

and "st' = st(|state.Gas := g', Storage := (Storage st) (Address ev := s)|)"

| (6) p x t g l t' g' s

where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st|)) (state.Gas st - costs (ASSIGN lv ex) ev cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"

and "lexp lv ev cd (st(|state.Gas := g|)) g = Normal((LStoreloc l, type.Storage t'),g)"

and "cps2mTypeCompatible t' (MTArray x t)"

and "cpm2s p l x t cd (Storage st (Address ev)) = Some s"

and "st' = st(|state.Gas := g', Storage := (Storage st) (Address ev := s)|)"

| (7) p x t g l t' g' m m'

where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st|)) (state.Gas st - costs (ASSIGN lv ex) ev cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"

and "lexp lv ev cd (st(|state.Gas := g|)) g = Normal((LMemloc l, type.Memory t'),g)"

and "(MTArray x t) = t'"

and "cpm2m p (ShowL_{nat} (Toploc (Memory st))) x t cd (snd (allocate (Memory st))) = Some m"

and "(updateStore l (MPointer (ShowL_{nat} (Toploc (Memory st)))) m) = m'"

and "st' = st(|state.Gas := g', Memory := m'|)"

```

| (8) p x t g l t' g'
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KMemptr p, type.Memory (MArray x t)), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LStackloc l, type.Memory t'),g)"
    and "t' = MArray x t"
    and "st' = st(|state.Gas := g', Stack := updateStore l (KMemptr p) (Stack st))"
| (9) p x t g l t' g' p' s
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KMemptr p, type.Memory (MArray x t)), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LStackloc l, type.Storage t'),g)"
    and "cps2mTypeCompatible t' (MArray x t)"
    and "accessStore l (Stack st) = Some (KStoptr p)"
    and "cpm2s p p' x t (Memory st) (Storage st (Address ev)) = Some s"
    and "st' = st(|state.Gas := g', Storage := (Storage st) (Address ev := s))"
| (10) p x t g l t' g' s
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KMemptr p, type.Memory (MArray x t)), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LStoreloc l, type.Storage t'),g)"
    and "cps2mTypeCompatible t' (MArray x t)"
    and "cpm2s p l x t (Memory st) (Storage st (Address ev)) = Some s"
    and "st' = st(|state.Gas := g', Storage := (Storage st) (Address ev := s))"
| (11) p x t g l t' g'
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KMemptr p, type.Memory (MArray x t)), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LMemloc l, type.Memory t'),g)"
    and "t' = MArray x t"
    and "st' = st(|state.Gas := g', Memory := updateStore l (MPointer p) (Memory st))"
| (12) p x t g l t' g' p' m st''
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KStoptr p, type.Storage (STArray x t)), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LStackloc l, type.Memory t'),g)"
    and "cps2mTypeCompatible (STArray x t) t'"
    and "accessStore l (Stack st) = Some (KMemptr p)"
    and "st'' = updateStore l (KMemptr (ShowLnat (Toploc (Memory st)))) (Stack st)"
    and "cps2m p (ShowLnat (Toploc (Memory st))) x t (Storage st (Address ev)) (snd (allocate(Memory
st))) = Some m"
    and "st' = st(|state.Gas := g', Memory := m, Stack := st'')"
| (13) p x t g l t' g'
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KStoptr p, type.Storage (STArray x t)), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LStackloc l, type.Storage t'),g)"
    and "t' = STArray x t"
    and "st' = st(|state.Gas := g', Stack := updateStore l (KStoptr p) (Stack st))"
| (14) p x t g l t' g' s
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KStoptr p, type.Storage (STArray x t)), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LStoreloc l, type.Storage t'),g)"
    and "t' = (STArray x t)"
    and "copy p l x t (Storage st (Address ev)) = Some s"
    and "st' = st(|state.Gas := g', Storage := (Storage st) (Address ev := s))"
| (15) p x t g l t' g' m m'
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KStoptr p, type.Storage (STArray x t)), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LMemloc l, type.Memory t'),g)"
    and "cps2mTypeCompatible (STArray x t) t'"
    and "cps2m p (ShowLnat (Toploc (Memory st))) x t (Storage st (Address ev)) (snd (allocate(Memory
st))) = Some m"
    and "updateStore l (MPointer ((ShowLnat (Toploc (Memory st)))) m = m'"
    and "st' = st(|state.Gas := g', Memory := m')"
| (16) p t t' g l g' t''
  where "expr ex ev cd (st(|state.Gas := state.Gas st - costs (ASSIGN lv ex) ev cd st)) (state.Gas
st - costs (ASSIGN lv ex) ev cd st) = Normal ((KStoptr p, type.Storage (STMap t t')), g)"
    and "lexp lv ev cd (st(|state.Gas := g)) g = Normal((LStackloc l, type.Storage t'),g)"
    and "t'' = (STMap t t'"

```

```

    and "st' = st(state.Gas := g', Stack := updateStore 1 (KStoptr p) (Stack st))"
  <proof>

```

lemma comp:

```

  assumes "stmt (COMP s1 s2) ev cd st = Normal (x, st'"
  obtains (1) st''
  where "state.Gas st > costs (COMP s1 s2) ev cd st"
    and "stmt s1 ev cd (st(state.Gas := state.Gas st - costs (COMP s1 s2) ev cd st)) = Normal((),
st'')"
    and "stmt s2 ev cd st'' = Normal((), st'"
  <proof>

```

lemma ite:

```

  assumes "stmt (ITE ex s1 s2) ev cd st = Normal (x, st'"
  obtains (True) g
  where "state.Gas st > costs (ITE ex s1 s2) ev cd st"
    and "expr ex ev cd (st(state.Gas := state.Gas st - costs (ITE ex s1 s2) ev cd st)) (state.Gas st -
costs (ITE ex s1 s2) ev cd st) = Normal((KValue (ShowLbool True), Value TBool), g)"
    and "stmt s1 ev cd (st(state.Gas := g)) = Normal((), st'"
  | (False) g
  where "state.Gas st > costs (ITE ex s1 s2) ev cd st"
    and "expr ex ev cd (st(state.Gas := state.Gas st - costs (ITE ex s1 s2) ev cd st)) (state.Gas st -
costs (ITE ex s1 s2) ev cd st) = Normal((KValue (ShowLbool False), Value TBool), g)"
    and "stmt s2 ev cd (st(state.Gas := g)) = Normal((), st'"
  <proof>

```

lemma while:

```

  assumes "stmt (WHILE ex s0) ev cd st = Normal (x, st'"
  obtains (True) g st''
  where "state.Gas st > costs (WHILE ex s0) ev cd st"
    and "expr ex ev cd (st(state.Gas := state.Gas st - costs (WHILE ex s0) ev cd st)) (state.Gas st -
costs (WHILE ex s0) ev cd st) = Normal((KValue (ShowLbool True), Value TBool), g)"
    and "stmt s0 ev cd (st(state.Gas := g)) = Normal((), st'"
    and "stmt (WHILE ex s0) ev cd st'' = Normal ((), st'"
  | (False) g
  where "state.Gas st > costs (WHILE ex s0) ev cd st"
    and "expr ex ev cd (st(state.Gas := state.Gas st - costs (WHILE ex s0) ev cd st)) (state.Gas st -
costs (WHILE ex s0) ev cd st) = Normal((KValue (ShowLbool False), Value TBool), g)"
    and "st' = st(state.Gas := g)"
  <proof>

```

lemma whileE:

```

  fixes st ex sm ev cd
  defines "nGas ≡ state.Gas st - costs (WHILE ex sm) ev cd st"
  assumes "stmt (WHILE ex sm) ev cd st = Exception e"
  obtains (Gas) "e = Gas"
    | (Err) "e = Err"
    | (Exp) "expr ex ev cd (st(Gas := nGas)) nGas = Exception e"
    | (Stm) g where "expr ex ev cd (st(Gas := nGas)) nGas = Normal ((KValue (ShowLbool True), Value
TBool), g)" and "stmt sm ev cd (st(Gas := g)) = Exception e"
    | (While) g st' where "state.Gas st > costs (WHILE ex sm) ev cd st" and "expr ex ev cd (st(Gas
:= nGas)) nGas = Normal ((KValue (ShowLbool True), Value TBool), g)" and "stmt sm ev cd (st(Gas := g))
= Normal ((), st'" and "local.stmt (WHILE ex sm) ev cd st' = Exception e"
  <proof>

```

lemma invoke:

```

  fixes ev
  defines "e' members ≡ ffold (init members) (emptyEnv (Address ev) (Contract ev) (Sender ev) (Svalue
ev)) (fndom members)"
  assumes "stmt (INVOKE i xe) ev cd st = Normal (x, st'"
  obtains ct fb fp f ei cdi ki mi g st''
  where "state.Gas st > costs (INVOKE i xe) ev cd st"
    and "ep $$ Contract ev = Some (ct, fb)"

```

```

    and "ct $$ i = Some (Method (fp, False, f))"
    and "load False fp xe (e' ct) emptyTypedStore emptyStore (Memory (st(|state.Gas := state.Gas st
- costs (INVOKE i xe) ev cd st))) ev cd (st(|state.Gas := state.Gas st - costs (INVOKE i xe) ev cd st))
(state.Gas st - costs (INVOKE i xe) ev cd st) = Normal ((el, cdl, kl, ml), g)"
    and "stmt f el cdl (st(|state.Gas:= g, Stack:=kl, Memory:=ml)) = Normal ((), st'')"
    and "st' = st''(|Stack:=Stack st)"
<proof>

```

lemma external:

```

fixes ev
defines "e' members adv c v ≡ ffold (init members) (emptyEnv adv c (Address ev) v) (fmdom members)"
assumes "stmt (EXTERNAL ad' i xe val) ev cd st = Normal (x, st')"
obtains (Some) adv c g ct cn fb' v t g' v' fp f el cdl kl ml g'' acc st''
  where "state.Gas st > costs (EXTERNAL ad' i xe val) ev cd st"
    and "expr ad' ev cd (st(|state.Gas := state.Gas st - costs (EXTERNAL ad' i xe val) ev cd st))
(state.Gas st - costs (EXTERNAL ad' i xe val) ev cd st) = Normal ((KValue adv, Value TAddr), g)"
    and "adv ≠ Address ev"
    and "Type (Accounts (st(|state.Gas := g)) adv) = Some (atype.Contract c)"
    and "ep $$ (c) = Some (ct, cn, fb')"
    and "expr val ev cd (st(|state.Gas := g)) g = Normal ((KValue v, Value t), g')"
    and "convert t (TUInt b256) v = Some v'"
    and "fmlookup ct i = Some (Method (fp, True, f))"
    and "load True fp xe (e' ct adv c v') emptyTypedStore emptyStore emptyTypedStore ev cd
(st(|state.Gas := g')) g' = Normal ((el, cdl, kl, ml), g'')"
    and "transfer (Address ev) adv v' (Accounts (st(|state.Gas := g''))) = Some acc"
    and "stmt f el cdl (st(|state.Gas := g'', Accounts := acc, Stack:=kl, Memory:=ml)) = Normal ((),
st'')"
    and "st' = st''(|Stack:=Stack st, Memory := Memory st)"
  | (None) adv c g ct cn fb' v t g' v' acc st''
  where "state.Gas st > costs (EXTERNAL ad' i xe val) ev cd st"
    and "expr ad' ev cd (st(|state.Gas := state.Gas st - costs (EXTERNAL ad' i xe val) ev cd st))
(state.Gas st - costs (EXTERNAL ad' i xe val) ev cd st) = Normal ((KValue adv, Value TAddr), g)"
    and "adv ≠ Address ev"
    and "Type (Accounts (st(|state.Gas := g)) adv) = Some (atype.Contract c)"
    and "ep $$ c = Some (ct, cn, fb')"
    and "expr val ev cd (st(|state.Gas := g)) g = Normal ((KValue v, Value t), g')"
    and "convert t (TUInt b256) v = Some v'"
    and "ct $$ i = None"
    and "transfer (Address ev) adv v' (Accounts st) = Some acc"
    and "stmt fb' (e' ct adv c v') emptyTypedStore (st(|state.Gas := g', Accounts := acc,
Stack:=emptyStore, Memory:=emptyTypedStore)) = Normal ((), st'')"
    and "st' = st''(|Stack:=Stack st, Memory := Memory st)"
<proof>

```

lemma transfer:

```

fixes ev
defines "e' members adv c st v ≡ ffold (init members) (emptyEnv adv c (Address ev) v) (fmdom
members)"
assumes "stmt (TRANSFER ad ex) ev cd st = Normal (x, st')"
obtains (Contract) v t g adv c g' v' acc ct cn f st''
  where "state.Gas st > costs (TRANSFER ad ex) ev cd st"
    and "expr ad ev cd (st(|state.Gas := state.Gas st - costs (TRANSFER ad ex) ev cd st)) (state.Gas
st - costs (TRANSFER ad ex) ev cd st) = Normal ((KValue adv, Value TAddr), g)"
    and "expr ex ev cd (st(|state.Gas := g)) g = Normal ((KValue v, Value t), g')"
    and "convert t (TUInt b256) v = Some v'"
    and "Type (Accounts (st(|state.Gas := g)) adv) = Some (atype.Contract c)"
    and "ep $$ c = Some (ct, cn, f)"
    and "transfer (Address ev) adv v' (Accounts st) = Some acc"
    and "stmt f (e' ct adv c (st(|state.Gas := g')) v') emptyTypedStore (st(|state.Gas := g', Accounts
:= acc, Stack:=emptyStore, Memory:=emptyTypedStore)) = Normal ((), st'')"
    and "st' = st''(|Stack:=Stack st, Memory := Memory st)"
  | (EOA) v t g adv g' v' acc
  where "state.Gas st > costs (TRANSFER ad ex) ev cd st"
    and "expr ad ev cd (st(|state.Gas := state.Gas st - costs (TRANSFER ad ex) ev cd st)) (state.Gas

```

5 Expressions and Statements

```

st - costs (TRANSFER ad ex) ev cd st = Normal ((KValue adv, Value TAddr), g)"
  and "expr ex ev cd (st(|state.Gas := g|)) g = Normal ((KValue v, Value t), g)"
  and "convert t (TUInt b256) v = Some v'"
  and "Type (Accounts (st(|state.Gas := g|)) adv) = Some EOA"
  and "transfer (Address ev) adv v' (Accounts st) = Some acc"
  and "st' = st(|state.Gas:=g', Accounts:=acc|)"

```

⟨proof⟩

lemma blockNone:

```

fixes ev
assumes "stmt (BLOCK (id0, tp, None) s) ev cd st = Normal (x, st)"
obtains cd' mem' sck' e'
  where "state.Gas st > costs (BLOCK (id0, tp, None) s) ev cd st"
    and "decl id0 tp None False cd (Memory (st(|state.Gas := state.Gas st - costs (BLOCK (id0, tp, None) s) ev cd st))) ((Storage (st(|state.Gas := state.Gas st - costs (BLOCK (id0, tp, None) s) ev cd st))) (Address ev)) (cd, Memory (st(|state.Gas := state.Gas st - costs (BLOCK (id0, tp, None) s) ev cd st)), Stack (st(|state.Gas := state.Gas st - costs (BLOCK (id0, tp, None) s) ev cd st)), ev) = Some (cd', mem', sck', e'"
    and "stmt s e' cd' (st(|state.Gas := state.Gas st - costs (BLOCK (id0, tp, None) s) ev cd st, Stack := sck', Memory := mem')) = Normal ((), st)"

```

⟨proof⟩

lemma blockSome:

```

fixes ev
assumes "stmt (BLOCK (id0, tp, Some ex') s) ev cd st = Normal (x, st)"
obtains v t g cd' mem' sck' e'
  where "state.Gas st > costs (BLOCK (id0, tp, Some ex') s) ev cd st"
    and "expr ex' ev cd (st(|state.Gas := state.Gas st - costs (BLOCK (id0, tp, Some ex') s) ev cd st))
(state.Gas st - costs (BLOCK (id0, tp, Some ex') s) ev cd st) = Normal((v,t),g)"
    and "decl id0 tp (Some (v, t)) False cd (Memory (st(|state.Gas := g|))) ((Storage (st(|state.Gas := g|))) (Address ev))
(cd, Memory (st(|state.Gas := g|)), Stack (st(|state.Gas := g|)), ev) = Some (cd', mem', sck', e'"
    and "stmt s e' cd' (st(|state.Gas := g, Stack := sck', Memory := mem')) = Normal ((), st)"

```

⟨proof⟩

lemma new:

```

fixes i xe val ev cd st
defines "st0 ≡ st(|state.Gas := state.Gas st - costs (NEW i xe val) ev cd st)"
defines "adv0 ≡ hash_version (Address ev) (ShowLnat (Contracts (Accounts st0 (Address ev))))"
defines "st1 g ≡ st(|state.Gas := g, Accounts := (Accounts st)(adv0 := (|Bal = ShowLint 0, Type = Some (atype.Contract i), Contracts = 0|)), Storage:=(Storage st)(adv0 := {$$}))"
defines "e' members c v ≡ ffold (init members) (emptyEnv adv0 c (Address ev) v) (fndom members)"
assumes "stmt (NEW i xe val) ev cd st = Normal (x, st)"
obtains v t g ct cn fb ei cdi kl ml g' acc st'' v'
  where "state.Gas st > costs (NEW i xe val) ev cd st"
    and "Type (Accounts st0 adv0) = None"
    and "expr val ev cd st0 (state.Gas st0) = Normal((KValue v, Value t),g)"
    and "convert t (TUInt b256) v = Some v'"
    and "ep $$ i = Some (ct, cn, fb)"
    and "load True (fst cn) xe (e' ct i v') emptyTypedStore emptyStore emptyTypedStore ev cd (st1 g)
g = Normal ((ei, cdi, kl, ml), g'"
    and "transfer (Address ev) adv0 v' (Accounts (st1 g')) = Some acc"
    and "stmt (snd cn) ei cdi (st1 g'(|Accounts := acc, Stack:=kl, Memory:=ml|)) = Normal ((), st'')"
    and "st' = incrementAccountContracts (Address ev) (st''(|Stack:=Stack st, Memory := Memory st))"

```

⟨proof⟩

lemma atype_same:

```

assumes "stmt stm ev cd st = Normal (x, st)"
  and "Type (Accounts st ad) = Some ctype"
shows "Type (Accounts st' ad) = Some ctype"

```

⟨proof⟩

```

declare lexp.simps[simp del, solidity_symbex add]
declare stmt.simps[simp del, solidity_symbex add]

end

```

5.3.6 A minimal cost model

```

fun costs_min :: "s ⇒ environment ⇒ calldataT ⇒ state ⇒ gas"
  where
    "costs_min SKIP e cd st = 0"
  | "costs_min (ASSIGN lv ex) e cd st = 0"
  | "costs_min (COMP s1 s2) e cd st = 0"
  | "costs_min (ITE ex s1 s2) e cd st = 0"
  | "costs_min (WHILE ex s0) e cd st = 1"
  | "costs_min (TRANSFER ad ex) e cd st = 1"
  | "costs_min (BLOCK (id0, tp, ex) s) e cd st = 0"
  | "costs_min (INVOKE _ _) e cd st = 1"
  | "costs_min (EXTERNAL _ _ _ _) e cd st = 1"
  | "costs_min (NEW _ _ _) e cd st = 1"

fun costs_ex :: "e ⇒ environment ⇒ calldataT ⇒ state ⇒ gas"
  where
    "costs_ex (e.INT _ _) e cd st = 0"
  | "costs_ex (e.UINT _ _) e cd st = 0"
  | "costs_ex (e.ADDRESS _) e cd st = 0"
  | "costs_ex (e.BALANCE _) e cd st = 0"
  | "costs_ex THIS e cd st = 0"
  | "costs_ex SENDER e cd st = 0"
  | "costs_ex VALUE e cd st = 0"
  | "costs_ex (TRUE) e cd st = 0"
  | "costs_ex (FALSE) e cd st = 0"
  | "costs_ex (LVAL _) e cd st = 0"
  | "costs_ex (PLUS _ _) e cd st = 0"
  | "costs_ex (MINUS _ _) e cd st = 0"
  | "costs_ex (EQUAL _ _) e cd st = 0"
  | "costs_ex (LESS _ _) e cd st = 0"
  | "costs_ex (AND _ _) e cd st = 0"
  | "costs_ex (OR _ _) e cd st = 0"
  | "costs_ex (NOT _) e cd st = 0"
  | "costs_ex (CALL _ _) e cd st = 1"
  | "costs_ex (ECALL _ _ _) e cd st = 1"
  | "costs_ex CONTRACTS e cd st = 0"

global_interpretation solidity: statement_with_gas costs_ex fmemory costs_min
  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⟩

```

5.4 Examples (Statements)

5.4.1 msel

```

abbreviation mmemory2::memoryT
  where "mmemory2 ≡
    (Mapping = fmap_of_list
      [(STR '3.2', MPointer STR '5')],
      Toploc = 1,
      Typed_Mapping = fmap_of_list

```

```
[(STR ''3.2'', MArray 6 (MValue TBool))]]"
```

```
lemma "msel True (MArray 5 (MArray 6 (MValue TBool))) (STR ''2'') [UINT b8 3] empty
emptyTypedStore (mystate(|state.Gas:=1|)) 1
= Normal ((STR ''3.2'', MArray 6 (MValue TBool)), 1)"
⟨proof⟩
```

```
lemma "msel True (MArray 5 (MArray 6 (MValue TBool))) (STR ''2'') [UINT b8 3, UINT b8 4] empty
emptyTypedStore (mystate(|state.Gas:=1,Memory:=mymemory2|)) 1
= Normal ((STR ''4.5'', MValue TBool), 1)" ⟨proof⟩
```

```
lemma "msel True (MArray 5 (MArray 6 (MValue TBool))) (STR ''2'') [UINT b8 5] empty
emptyTypedStore (mystate(|state.Gas:=1,Memory:=mymemory2|)) 1
= Exception (Err)" ⟨proof⟩
```

```
end
```

5.5 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
  add:string_literal_opt)+)

declare Let_def [solidity_symbex]
      o_def [solidity_symbex]

end
```

6.2 Solidity 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

Generalized Unit Tests lemma "createSInt b8 500 = STR '-12'" <proof>

lemma "STR '-92134039538802366542421159375273829975'"
  = createSInt b128 45648483135649456465465452123894894554654654654646999465"
  <proof>
```

```

lemma "STR ''-128'' = createSInt b8 (-128)" <proof>
lemma "STR ''244'' = createUInt b8 500" <proof>
lemma "STR ''220443428915524155977936330922349307608''
      = createUInt b128 4564848313564945646546545212389489455465465465464699946544654654654168"
      <proof>
lemma "less (TUInt b144) (TSInt b160) (STR ''5'') (STR ''8'') = Some(STR ''True'', TBool) "
      <proof>

```

6.2.1 Code Generator Setup and Local Tests

Utils

```

definition EMPTY::"String.literal" where "EMPTY = STR ''''"
definition FAILURE::"String.literal" where "FAILURE = STR ''Failure''"
fun intersperse :: "String.literal ⇒ String.literal list ⇒ String.literal" where
  "intersperse s [] = EMPTY"
| "intersperse s [x] = x"
| "intersperse s (x # xs) = x + s + intersperse s xs"
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 ⇒ '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)"
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"

```

Valuetypes

```

fun dumpVvaluetypes::"types ⇒ valuetype ⇒ String.literal" where
  "dumpVvaluetypes (TSInt _) n = n"

```

```

| "dumpValuetypes (TUInt _) n = n"
| "dumpValuetypes TBool b = (if b = (STR ''True'') then STR ''true'' else STR ''false'')"
| "dumpValuetypes TAddr ad = ad"

```

Memory

```

datatype dataMemory = MArray "dataMemory list"
  | MBool bool
  | MInt int
  | MAddress address

```

```

fun loadRecMemory :: "location ⇒ dataMemory ⇒ memoryT ⇒ memoryT" and
  iterateM :: "location ⇒ memoryT × nat ⇒ dataMemory ⇒ memoryT × nat" where
  "loadRecMemory loc (MArray dat) mem = fst (foldl (iterateM loc) (updateStore loc (MPointer loc)
mem,0) dat)"
| "loadRecMemory loc (MBool b) mem = updateStore loc ((MValue o ShowLbool) b) mem "
| "loadRecMemory loc (MInt i) mem = updateStore loc ((MValue o ShowLint) i) mem "
| "loadRecMemory loc (MAddress ad) mem = updateStore loc (MValue ad) mem"
| "iterateM loc (mem,x) d = (loadRecMemory (hash loc (ShowLnat x)) d mem, Suc x)"

```

```

definition loadMemory :: "dataMemory list ⇒ memoryT ⇒ memoryT" where
"loadMemory dat mem = (let loc = ShowLnat (Toploc mem);
  (m, _) = foldl (iterateM loc) (mem, 0) dat
  in (snd o allocate) m)"

```

```

fun dumprecMemory :: "location ⇒ mtypes ⇒ memoryT ⇒ String.literal ⇒ String.literal ⇒
String.literal" where
"dumprecMemory loc tp mem ls str =
  (case accessStore loc mem of
    Some (MPointer l) ⇒
      (case tp of
        (MArray x t) ⇒ iter (λi str' . dumprecMemory ((hash l o ShowLnat) i) t mem
          (ls + (STR ''['') + (ShowLnat i) + (STR ''']''))) str') str x
        | _ ⇒ FAILURE)
    | Some (MValue v) ⇒
      (case tp of
        MValue t ⇒ str + ls + (STR ''=='') + dumpValuetypes t v + (STR ''↔'')
        | _ ⇒ FAILURE)
    | None ⇒ FAILURE)"

```

```

definition dumpMemory :: "location ⇒ nat ⇒ mtypes ⇒ memoryT ⇒ String.literal ⇒String.literal
⇒String.literal" where
"dumpMemory loc x t mem ls str = iter (λi. dumprecMemory ((hash loc (ShowLnat i))) t mem (ls + STR
''['' + (ShowLnat i + STR ''']''))) str x"

```

Storage

```

datatype dataStorage =
  SArray "dataStorage list"
  | SMap "(String.literal × dataStorage) list"
  | SBool bool
  | SInt int
  | SAddress address

```

```

fun goStorage :: "location ⇒ (String.literal × stypes) ⇒ (String.literal × stypes)" where
  "goStorage l (s, STArray _ t) = (s + (STR ''['') + (convert l) + (STR ''']''), t)"
| "goStorage l (s, SMap _ t) = (s + (STR ''['') + (convert l) + (STR ''']''), t)"
| "goStorage l (s, STValue t) = (s + (STR ''['') + (convert l) + (STR ''']''), STValue t)"

```

```

fun dumpSingleStorage :: "storageT ⇒ String.literal ⇒ stypes ⇒ (location × location) ⇒
String.literal ⇒ String.literal" where
"dumpSingleStorage sto id' tp (loc,l) str =
  (case foldr goStorage (tolist loc) (str + id', tp) of
    (s, STValue t) ⇒

```

```

    (case sto $$ (loc + 1) of
      Some v ⇒ s + (STR '==') + dumpValuetypes t v
    | None ⇒ FAILURE)
  | _ ⇒ FAILURE)"

```

definition `iterate where`

```

"iterate loc t id' sto s l = dumpSingleStorage sto id' t (splitAt (length (String.explode l) - length
(String.explode loc) - 1) 1) s + (STR '↔')"
```

fun `dumpStorage :: "storageT ⇒ location ⇒ String.literal ⇒ stypes ⇒ String.literal ⇒ String.literal"` **where**

```

"dumpStorage sto loc id' (STArray _ t) str = foldl (iterate loc t id' sto) str (locations_map loc
sto)"
| "dumpStorage sto loc id' (STMap _ t) str = foldl (iterate loc t id' sto) str (locations_map loc sto)"
| "dumpStorage sto loc id' (STValue t) str =
  (case sto $$ loc of
    Some v ⇒ str + id' + (STR '==') + dumpValuetypes t v + (STR '↔')
  | _ ⇒ str)"

```

fun `loadRecStorage :: "location ⇒ dataStorage ⇒ storageT ⇒ storageT"` **and**

`iterateSA :: "location ⇒ String.literal × nat ⇒ dataStorage ⇒ storageT × nat"` **and**

`iterateSM :: "location ⇒ String.literal × dataStorage ⇒ storageT ⇒ storageT"` **where**

```

"loadRecStorage loc (SArray dat) sto = fst (foldl (iterateSA loc) (sto,0) dat)"
| "loadRecStorage loc (SMap dat) sto = foldr (iterateSM loc) dat sto"
| "loadRecStorage loc (SBool b) sto = fmupd loc (ShowLbool b) sto"
| "loadRecStorage loc (SInt i) sto = fmupd loc (ShowLint i) sto"
| "loadRecStorage loc (SAddress ad) sto = fmupd loc ad sto"
| "iterateSA loc (s', x) d = (loadRecStorage (hash loc (ShowLnat x)) d s', Suc x)"
| "iterateSM loc (k, v) s' = loadRecStorage (hash loc k) v s'"

```

environment

datatype `dataEnvironment =`

```

Memarr "dataMemory list" |
CDarr "dataMemory list" |
Stoarr "dataStorage list" |
Stomap "(String.literal × dataStorage) list" |
Stackbool bool |
Stobool bool |
Stackint int |
Stoint int |
Stackaddr address |
Stoaddr address

```

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))"`

fun `loadsimpleEnvironment :: "(stack × calldataT × memoryT × storageT × environment) ⇒ (Identifier × type × dataEnvironment) ⇒ (stack × calldataT × memoryT × storageT × environment)"`

where

```

"loadsimpleEnvironment (k, c, m, s, e) (id', tp, d) = (case d of
  Stackbool b ⇒
    let (k', e') = astore id' tp (KValue (ShowLbool b)) (k, e)
    in (k', c, m, s, e')
| Stobool b ⇒
    let (s', e') = astore id' tp (ShowLbool b) (s, e)
    in (k, c, m, s', e')
| Stackint n ⇒
    let (k', e') = astore id' tp (KValue (ShowLint n)) (k, e)
    in (k', c, m, s, e')
| Stoint n ⇒
    let (s', e') = astore id' tp (ShowLint 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 = ShowLnat (Toploc c);
      c' = loadMemory a c;
      (k', e') = astack id' tp (KCDptr l) (k, e)
  in (k', c', m, s, e')
| Memarr a ⇒
  let l = ShowLnat (Toploc m);
      m' = loadMemory a m;
      (k', e') = astack id' tp (KMemptr l) (k, e)
  in (k', c, m', s, e')
| Stoarr a ⇒
  let s' = loadRecStorage id' (SArray a) s;
      e' = updateEnv id' tp (Storeloc id') e
  in (k, c, m, s', e')
| Stomap mp ⇒
  let s' = loadRecStorage id' (SMap mp) s;
      e' = updateEnv id' tp (Storeloc id') e
  in (k, c, m, s', e')
)"

```

```

definition getValueEnvironment :: "stack ⇒ calldataT ⇒ memoryT ⇒ storageT ⇒ environment ⇒
Identifier ⇒ String.literal ⇒ String.literal"

```

```

where
"getValueEnvironment 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 '==') + dumpValuetypes t v + (STR '↔'))
      | _ ⇒ FAILURE)
    | Some (KCDptr p) ⇒ (case tp of
      Calldata (MArray x t) ⇒ dumpMemory p x t c i txt
      | _ ⇒ FAILURE)
    | Some (KMemptr p) ⇒ (case tp of
      type.Memory (MArray x t) ⇒ dumpMemory p x t m i txt
      | _ ⇒ FAILURE)
    | Some (KStoptr p) ⇒ (case tp of
      type.Storage t ⇒ dumpStorage s p i t txt
      | _ ⇒ FAILURE))
    | Some (type.Storage t, Storeloc l) ⇒ dumpStorage s l i t txt
    | _ ⇒ FAILURE)
)"

```

```

definition dumpEnvironment :: "stack ⇒ calldataT ⇒ memoryT ⇒ storageT ⇒ environment ⇒ Identifier
list ⇒ String.literal"

```

```

where "dumpEnvironment k c m s e sl = foldr (getValueEnvironment k c m s e) sl EMPTY"

```

accounts

```

fun loadAccounts :: "accounts ⇒ (address × balance × atype × nat) list ⇒ accounts" where
  "loadAccounts acc [] = acc"
  | "loadAccounts acc ((ad, b, t, c)#as) = loadAccounts (acc (ad:=(Bal=b, Type=Some t, Contracts=c))) as"

```

```

fun dumpStorage::"storageT ⇒ (Identifier × member) ⇒ String.literal" where
  "dumpStorage s (i, Var x) = dumpStorage s i x EMPTY"
  | "dumpStorage s (i, Function x) = FAILURE"
  | "dumpStorage s (i, Method x) = FAILURE"

```

```

fun dumpMembers :: "atype option ⇒ environmentp ⇒ storageT ⇒ String.literal" where
  "dumpMembers None ep s = FAILURE"

```

```

| "dumpMembers (Some EOA) _ _ = STR ''EOA''"
| "dumpMembers (Some (atype.Contract name)) ep s =
  (case ep $$ name of
    Some (ct, _) => name + STR ''('' + (intersperse (STR ''','') (map (dumpStorage s) (filter (is_Var
o snd) (sorted_list_of_fmap ct)))) + STR '''))''
  | None => FAILURE)"

```

```

fun dumpAccount :: "nat => environment_p => state => address => String.literal"
where "dumpAccount 0 _ _ _ = FAILURE"
| "dumpAccount (Suc c) ep st a = a + STR ''': '' +
  STR ''balance=='' + Bal (Accounts st a) +
  STR '' - '' + dumpMembers (Type (Accounts st a)) ep (Storage st a) +
  iter (λx s. s + STR ''↔'' + dumpAccount c ep st (hash_version a (ShowL_nat x))) EMPTY
((Contracts (Accounts st a)))"

```

```

definition dumpAccounts :: "environment_p => state => address list => String.literal"
where "dumpAccounts ep st al = intersperse (STR ''↔'' (map (dumpAccount 1000 ep st) al))"

```

```

definition initAccount :: "(address × balance × atype × nat) list => accounts" where
"initAccount = loadAccounts emptyAccount"

```

```

type_synonym data_P = "(Identifier × member) list × ((Identifier × type) list × s) × s"

```

```

fun loadProc :: "Identifier => data_P => environment_p => environment_p"
where "loadProc i (xs, fb) = fmupd i (fmap_of_list xs, fb)"

```

6.2.2 Test Setup

```

definition (in statement_with_gas) eval :: "gas => s => address => Identifier => address => valuetype =>
(address × balance × atype × nat) list
=> (String.literal × type × data_Environment) list
=> String.literal"
where "eval g stm addr name adest aval acc dat
= (let (k,c,m,s,e) = foldl loadsimple_Environment (emptyStore, emptyTypedStore, emptyTypedStore,
fmempty, emptyEnv addr name adest aval) dat;
  a = initAccount acc;
  s' = emptyStorage (addr := s);
  z = (Accounts=a, Stack=k, Memory=m, Storage=s', Gas=g)
in (
  case (stmt stm e c z) of
    Normal ((), z') => (dump_Environment (Stack z') c (Memory z') (Storage z' addr) e (map (λ
(a,b,c). a) dat))
      + (dumpAccounts ep z' (map fst acc))
  | Exception Err => STR ''Exception''
  | Exception gas => STR ''OutOfGas''))"

```

```

global_interpretation soliditytest0: statement_with_gas costs_ex "fmap_of_list []" costs_min

```

```

defines stmt0 = "soliditytest0.stmt"
and lexp0 = soliditytest0.lexp
and expr0 = soliditytest0.expr
and ssel0 = soliditytest0.ssel
and rexp0 = soliditytest0.rexp
and msel0 = soliditytest0.msel
and load0 = soliditytest0.load
and eval0 = soliditytest0.eval
⟨proof⟩

```

```

lemma "eval0 1000
  SKIP
  (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
  EMPTY
  EMPTY
  (STR ''0'')

```

```

    [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'', EOA, 0)]
    [(STR ''v1'', (Value TBool, Stackbool True))]
    = STR ''v1==true[↔]089Be5381FcEa58aF334101414c04F993947C733: balance==100 -
EOA[↔]115f6e2F70210C14f7DB1AC69737a3CC78435d49: balance==100 - EOA''
    <proof>

```

lemma "eval0 1000

```

    SKIP
    (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
    EMPTY
    EMPTY
    (STR ''0'')
    [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'', EOA, 0)]
    [(STR ''v1'', (type.Memory (MArray 5 (MValue 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
balance==100 - EOA[↔]115f6e2F70210C14f7DB1AC69737a3CC78435d49: balance==100 - EOA''
    <proof>

```

lemma "eval0 1000

```

    (ITE FALSE (ASSIGN (Id (STR ''x'')) TRUE) (ASSIGN (Id (STR ''y'')) TRUE))
    (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
    EMPTY
    EMPTY
    (STR ''0'')
    [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'', EOA, 0)]
    [(STR ''x'', (Value TBool, Stackbool False)), (STR ''y'', (Value TBool, Stackbool False))]
    = STR ''y==true[↔]x==false[↔]089Be5381FcEa58aF334101414c04F993947C733: balance==100 -
EOA[↔]115f6e2F70210C14f7DB1AC69737a3CC78435d49: balance==100 - EOA''
    <proof>

```

lemma "eval0 1000

```

    (BLOCK (STR ''v2'', Value TBool, None) (ASSIGN (Id (STR ''v1'')) (LVAL (Id (STR ''v2'')))))
    (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
    EMPTY
    EMPTY
    (STR ''0'')
    [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'', EOA, 0)]
    [(STR ''v1'', (Value TBool, Stackbool True))]
    = STR ''v1==false[↔]089Be5381FcEa58aF334101414c04F993947C733: balance==100 -
EOA[↔]115f6e2F70210C14f7DB1AC69737a3CC78435d49: balance==100 - EOA''
    <proof>

```

lemma "eval0 1000

```

    (ASSIGN (Id (STR ''a_s120_21_m8'')) (LVAL (Id (STR ''a_s120_21_s8''))))
    (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
    EMPTY
    EMPTY
    (STR ''0'')
    [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0)]
    [(STR ''a_s120_21_s8'', type.Storage (SArray 1 (SArray 2 (SValue (TSInt b120))))),
Stoarr [SArray [SInt 347104507864064359095275590289383142, SInt 565831699297331399489670920129618233]]],
    ((STR ''a_s120_21_m8'', type.Memory (MArray 1 (MArray 2 (MValue (TSInt b120))))), Memarr
[MArray [MInt (290845675805142398428016622247257774), MInt ((-96834026877269277170645294669272226))]])]
    = STR ''a_s120_21_m8[0][0]==347104507864064359095275590289383142[↔]a_s120_21_m8[0][1]==56583169929733139948967092
balance==100 - EOA''
    <proof>

```

lemma "eval0 1000

```

    (ASSIGN (Ref (STR ''a_s8_32_m0'') [UINT b8 1]) (LVAL (Ref (STR ''a_s8_31_s7'') [UINT b8
0])))
    (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
    EMPTY
    EMPTY
    (STR ''0'')
    [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0)]
    [(STR ''a_s8_31_s7'', type.Storage (STArray 1 (STArray 3 (STValue (TSInt b8))))), Stoarr
[SArray [SInt ((98)), SInt ((-23)), SInt (36)]]],
    (STR ''a_s8_32_m0'', type.Memory (MArray 2 (MArray 3 (MValue (TSInt b8))))), 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_s8_32_m0[1][1]==100 - EOA''
    <proof>

```

lemma "eval0 1000

```

    SKIP
    (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
    EMPTY
    EMPTY
    (STR ''0'')
    [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR
''115f6e2F70210C14f7DB1AC69737a3CC78435d49'', STR ''100'', EOA, 0)]
    [(STR ''v1'', type.Storage (STMap (TUInt b32) (STValue (TUInt b8))), Stomap [(STR
''2129136830'', SInt 247)]]]
    = STR ''v1[2129136830]==247[↔]089Be5381FcEa58aF334101414c04F993947C733: balance==100 -
EOA[↔]115f6e2F70210C14f7DB1AC69737a3CC78435d49: balance==100 - EOA''
    <proof>

```

definition "testenv1 ≡ loadProc (STR ''mycontract'')

```

    ([ (STR ''v1'', Var (STValue TBool)),
    (STR ''m1'', Method ([, True, (ASSIGN (Id (STR ''v1'')) FALSE))],
    ([, SKIP),
    SKIP)
    fmempty"

```

global_interpretation soliditytest1: statement_with_gas costs_ex testenv1 costs_min

```

    defines stmt1 = "soliditytest1.stmt"
    and lexp1 = soliditytest1.lexp
    and expr1 = soliditytest1.expr
    and ssel1 = soliditytest1.ssel
    and rexp1 = soliditytest1.rexp
    and msel1 = soliditytest1.msel
    and load1 = soliditytest1.load
    and eval1 = soliditytest1.eval
    <proof>

```

lemma "eval1 1000

```

    (EXTERNAL (ADDRESS (STR ''myaddr'')) (STR ''m1'') [] (UINT b256 0))
    (STR ''local'')
    EMPTY
    (STR ''mycontract'')
    (STR ''0'')
    [(STR ''local'', STR ''100'', EOA, 0), (STR ''myaddr'', STR ''100'', atype.Contract (STR
''mycontract''), 0)]
    []
    = STR ''local: balance==100 - EOA[↔]myaddr: balance==100 - mycontract(v1==false[↔])''
    <proof>

```

lemma "eval1 1000

```

    (NEW (STR ''mycontract'') [] (UINT b256 10))
    (STR ''local'')
    EMPTY

```

```

    (STR ''mycontract'')
    (STR ''0'')
    [(STR ''local'', STR ''100'', EOA, 0), (STR ''myaddr'', STR ''100'', atype.Contract (STR
''mycontract''), 0)]
    []
    = STR ''local: balance==90 - EOA[←]0-local: balance==10 - mycontract()[←]myaddr: balance==100
- mycontract()'')
    <proof>

```

```

lemma "eval1 1000
  (
    COMP
    (NEW (STR ''mycontract'') [] (UINT b256 10))
    (EXTERNAL CONTRACTS (STR ''m1'') [] (UINT b256 0))
  )
  (STR ''local'')
  EMPTY
  (STR ''mycontract'')
  (STR ''0'')
  [(STR ''local'', STR ''100'', EOA, 0), (STR ''myaddr'', STR ''100'', atype.Contract (STR
''mycontract''), 0)]
  []
  = STR ''local: balance==90 - EOA[←]0-local: balance==10 - mycontract(v1==false[←])[←]myaddr:
balance==100 - mycontract()'')
  <proof>

```

```

definition "testenv2 ≡ loadProc (STR ''mycontract'')
  [(STR ''m1'', Function ([, False, UINT b8 5])),
  ([, SKIP),
  SKIP)
  fmemory"

```

```

global_interpretation soliditytest2: statement_with_gas costs_ex testenv2 costs_min
  defines stmt2 = "soliditytest2.stmt"
  and lexp2 = soliditytest2.lexp
  and expr2 = soliditytest2.expr
  and ssel2 = soliditytest2.ssel
  and rexp2 = soliditytest2.rexp
  and msel2 = soliditytest2.msel
  and load2 = soliditytest2.load
  and eval2 = soliditytest2.eval
  <proof>

```

```

lemma "eval2 1000
  (ASSIGN (Id (STR ''v1'')) (CALL (STR ''m1'') []))
  (STR ''myaddr'')
  (STR ''mycontract'')
  EMPTY
  (STR ''0'')
  [(STR ''myaddr'', STR ''100'', EOA, 0)]
  [(STR ''v1'', (Value (TUInt b8), Stackint 0))]
  = STR ''v1==5[←]myaddr: balance==100 - EOA''
  <proof>

```

```

definition "testenv3 ≡ loadProc (STR ''mycontract'')
  [(STR ''m1'',
  Function [(STR ''v2'', Value (TSInt b8)), (STR ''v3'', Value (TSInt b8))],
  False,
  PLUS (LVAL (Id (STR ''v2''))) (LVAL (Id (STR ''v3''))))],
  ([, SKIP),
  SKIP)
  fmemory"

```

```

global_interpretation soliditytest3: statement_with_gas costs_ex testenv3 costs_min

```

```

defines stmt3 = "soliditytest3.stmt"
  and lexp3 = soliditytest3.lexp
  and expr3 = soliditytest3.expr
  and ssel3 = soliditytest3.ssel
  and rexp3 = soliditytest3.rexp
  and msel3 = soliditytest3.msel
  and load3 = soliditytest3.load
  and eval3 = soliditytest3.eval
  ⟨proof⟩

lemma "eval3 1000
  (ASSIGN (Id (STR ''v1'')) (CALL (STR ''m1'') [e.INT b8 3, e.INT b8 4]))
  (STR ''myaddr'')
  (STR ''mycontract'')
  EMPTY
  (STR ''0'')
  [(STR ''myaddr'', STR ''100'', EOA, 0), (STR ''mya'', STR ''100'', EOA, 0)]
  [(STR ''v1'', (Value (TSInt b8), Stackint 0))]
  = STR ''v1==7[↔]myaddr: balance==100 - EOA[↔]mya: balance==100 - EOA''
  ⟨proof⟩

definition "testenv4 ≡ loadProc (STR ''mycontract'')
  ([ (STR ''m1'', Function ([ (STR ''v2'', Value (TSInt b8)), (STR ''v3'', Value (TSInt b8))],
  True, PLUS (LVAL (Id (STR ''v2''))) (LVAL (Id (STR ''v3'')))))]),
  ([], SKIP),
  SKIP)
  fmemory"

global_interpretation soliditytest4: statement_with_gas costs_ex testenv4 costs_min
  defines stmt4 = "soliditytest4.stmt"
  and lexp4 = soliditytest4.lexp
  and expr4 = soliditytest4.expr
  and ssel4 = soliditytest4.ssel
  and rexp4 = soliditytest4.rexp
  and msel4 = soliditytest4.msel
  and load4 = soliditytest4.load
  and eval4 = soliditytest4.eval
  ⟨proof⟩

lemma "eval4 1000
  (ASSIGN (Id (STR ''v1'')) (ECALL (ADDRESS (STR ''extaddr'')) (STR ''m1'') [e.INT b8 3, e.INT
  b8 4]))
  (STR ''myaddr'')
  EMPTY
  EMPTY
  (STR ''0'')
  [(STR ''myaddr'', STR ''100'', EOA, 0), (STR ''extaddr'', STR ''100'', atype.Contract (STR
  ''mycontract''), 0)]
  [(STR ''v1'', (Value (TSInt b8), Stackint 0))]
  = STR ''v1==7[↔]myaddr: balance==100 - EOA[↔]extaddr: balance==100 - mycontract()''
  ⟨proof⟩

definition "testenv5 ≡ loadProc (STR ''mycontract'')
  ([], ([], SKIP), SKIP)
  fmemory"

global_interpretation soliditytest5: statement_with_gas costs_ex testenv5 costs_min
  defines stmt5 = "soliditytest5.stmt"
  and lexp5 = soliditytest5.lexp
  and expr5 = soliditytest5.expr
  and ssel5 = soliditytest5.ssel
  and rexp5 = soliditytest5.rexp
  and msel5 = soliditytest5.msel
  and load5 = soliditytest5.load

```

```

    and eval5 = soliditytest5.eval
  <proof>

```

```

lemma "eval5 1000
  (TRANSFER (ADDRESS (STR ''myaddr'')) (UINT b256 10))
  (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
  EMPTY
  EMPTY
  (STR ''0'')
  [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR ''myaddr'',
STR ''100'', atype.Contract (STR ''mycontract''), 0)]
  []
  = STR ''089Be5381FcEa58aF334101414c04F993947C733: balance==90 - EOA[↔]myaddr: balance==110 -
mycontract()'')
  <proof>

```

```

definition "testenv6 ≡ loadProc (STR ''Receiver'')
  ([(STR ''hello'', Var (STValue TBool))],
  [], SKIP),
  ASSIGN (Id (STR ''hello'')) TRUE)
  fmemory"

```

```

global_interpretation soliditytest6: statement_with_gas costs_ex testenv6 costs_min
  defines stmt6 = "soliditytest6.stmt"
  and lexp6 = soliditytest6.lexp
  and expr6 = soliditytest6.expr
  and ssel6 = soliditytest6.ssel
  and rexp6 = soliditytest6.rexp
  and msel6 = soliditytest6.msel
  and load6 = soliditytest6.load
  and eval6 = soliditytest6.eval
  <proof>

```

```

lemma "eval6 1000
  (TRANSFER (ADDRESS (STR ''ReceiverAd'')) (UINT b256 10))
  (STR ''SenderAd'')
  EMPTY
  EMPTY
  (STR ''0'')
  [(STR ''ReceiverAd'', STR ''100'', atype.Contract (STR ''Receiver''), 0), (STR ''SenderAd'',
STR ''100'', EOA, 0)]
  []
  = STR ''ReceiverAd: balance==110 - Receiver(hello==true[↔])[↔]SenderAd: balance==90 - EOA''
  <proof>

```

```

definition "testenv7 ≡ loadProc (STR ''mycontract'')
  ( [], ( [], SKIP), SKIP)
  fmemory"

```

```

global_interpretation soliditytest7: statement_with_gas costs_ex testenv7 costs_min
  defines stmt7 = "soliditytest7.stmt"
  and lexp7 = soliditytest7.lexp
  and expr7 = soliditytest7.expr
  and ssel7 = soliditytest7.ssel
  and rexp7 = soliditytest7.rexp
  and msel7 = soliditytest7.msel
  and load7 = soliditytest7.load
  and eval7 = soliditytest7.eval
  <proof>

```

```

lemma "eval7 1000
  (COMP(COMP(((ASSIGN (Id (STR ''x'')) (e.UINT b8 0))))(TRANSFER (ADDRESS (STR ''myaddr''))
(UINT b256 5)))(SKIP))
  (STR ''089Be5381FcEa58aF334101414c04F993947C733'')

```

```

EMPTY
EMPTY
(STR ''0'')
[(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR ''myaddr'',
STR ''100'', atype.Contract (STR ''mycontract''),0)]
[(STR ''x'', (Value (TUInt b8), Stackint 9))]
= STR ''x==0 [↔] 089Be5381FcEa58aF334101414c04F993947C733: balance==95 - EOA [↔] myaddr:
balance==105 - mycontract()'']
⟨proof⟩

```

6.2.3 The Final Code Export

```

consts ReadLS  :: "String.literal ⇒ s"
consts ReadLacc :: "String.literal ⇒ (String.literal × String.literal × atype × nat) list"
consts ReadLdat :: "String.literal ⇒ (String.literal × type × dataEnvironment) list"
consts ReadLP  :: "String.literal ⇒ dataP list"

code_printing
  constant ReadLS  → (Haskell) "Prelude.read"
  | constant ReadLacc → (Haskell) "Prelude.read"
  | constant ReadLdat → (Haskell) "Prelude.read"
  | constant ReadLP  → (Haskell) "Prelude.read"

fun main_stub :: "String.literal list ⇒ (int × String.literal)"
  where
    "main_stub [stm, saddr, name, raddr, val, acc, dat]
      = (0, eval0 1000 (ReadLS stm) saddr name raddr val (ReadLacc acc) (ReadLdat 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 eval0 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

Simple Examples

```

lemma "msel True (MArray 5 (MArray 6 (MValue TBool))) (STR ''2'') [UINT b8 3] empty
emptyTypedStore (mystate(|state.Gas:=1|)) 1
= Normal ((STR ''3.2'', MArray 6 (MValue TBool)), 1)" ⟨proof⟩

```

```

lemma "msel True (MArray 5 (MArray 6 (MValue TBool))) (STR ''2'') [UINT b8 3, UINT b8 4] empty
emptyTypedStore (mystate(|state.Gas:=1,Memory:=mymemory2|)) 1
= Normal ((STR ''4.5'', MValue TBool), 1)" ⟨proof⟩

```

```

lemma "msel True (MArray 5 (MArray 6 (MValue TBool))) (STR ''2'') [UINT b8 5] empty

```

```
emptyTypedStore (mystate(|state.Gas:=1,Memory:=mymemory2|)) 1
= Exception (Err)" <proof>
```

A More Complex Example Including Memory Copy

```
abbreviation P1::s
  where "P1 ≡ COMP (ASSIGN (Id (STR ''sa'')) (LVAL (Id (STR ''ma''))))
          (ASSIGN (Ref (STR ''sa'')) [UINT b8 0]) TRUE)"

abbreviation myenv::environment
  where "myenv ≡ updateEnv (STR ''ma'') (type.Memory (MArray 1 (MValue TBool))) (Stackloc (STR
''1''))
          (updateEnv (STR ''sa'') (type.Storage (STArray 1 (STValue TBool))) (Storeloc (STR
''1''))
          (emptyEnv (STR ''ad'') EMPTY (STR ''ad'') (STR ''0'')))"

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

abbreviation mystore::"address ⇒ storageT"
  where "mystore ≡ λ _ . fmemory"

abbreviation mymemory::memoryT
  where "mymemory ≡ updateTypeStore (STR ''0.1'') (MValue TBool) (updateStore (STR ''0.1'') (MValue
(STR ''False'')) emptyTypedStore)"

abbreviation mystorage::storageT
  where "mystorage ≡ fmupd (STR ''0.1'') (STR ''True'') fmemory"

declare [[ML_print_depth = 10000]]
value <(stmt P1 myenv emptyTypedStore (|Accounts=emptyAccount, Stack=mystack, Memory=mymemory,
Storage=(mystore ((STR ''ad''):= mystorage)), state.Gas=1000|)>

lemma <(stmt P1 myenv emptyTypedStore (|Accounts=emptyAccount, Stack=mystack, Memory=mymemory,
Storage=(mystore ((STR ''ad''):= mystorage)), state.Gas=1000|)
  = Normal ((), (|Accounts = emptyAccount, Stack = (|Mapping = fmap_of_list [(STR ''1''), KMemptr STR
''1'']), Toploc = 0|),
          Memory = (|Mapping = fmap_of_list [(STR ''0.1''), MValue STR ''False'']), Toploc =
0, Typed_Mapping=fmap_of_list [(STR ''0.1''), MValue TBool]), Storage = (mystore ((STR ''ad''):=
mystorage)), state.Gas = 1000|) >
  <proof>

end
```

6.3 Generating an Executable of the Evaluator (Compile_Evaluator)

```
theory
  Compile_Evaluator
  imports
    Solidity_Evaluator
begin

compile_generated_files _ (in Solidity_Evaluator) export_files "solidity-evaluator" (exe)
  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 #> unuffix ".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 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 . ."]])
in
  writeln (res)
end>

end

```

7 Verification Support

This chapter presents a weakest precondition calculus and corresponding verification condition generator.

```
theory Weakest_Precondition
  imports Solidity_Main
begin
```

7.1 Setup for Monad VCG (Weakest_Precondition)

```
lemma wpstackvalue[wprule]:
  assumes " $\bigwedge v. a = KValue\ v \implies wp\ (f\ v)\ P\ E\ s$ "
    and " $\bigwedge p. a = KCDptr\ p \implies wp\ (g\ p)\ P\ E\ s$ "
    and " $\bigwedge p. a = KMemptr\ p \implies wp\ (h\ p)\ P\ E\ s$ "
    and " $\bigwedge p. a = KStoptr\ p \implies wp\ (i\ p)\ P\ E\ s$ "
  shows " $wp\ (case\ a\ of\ KValue\ v \Rightarrow f\ v\ | KCDptr\ p \Rightarrow g\ p\ | KMemptr\ p \Rightarrow h\ p\ | KStoptr\ p \Rightarrow i\ p)\ P\ E\ s$ "
  <proof>
```

```
lemma wpmtypes[wprule]:
  assumes " $\bigwedge i\ m. a = MArray\ i\ m \implies wp\ (f\ i\ m)\ P\ E\ s$ "
    and " $\bigwedge t. a = MValue\ t \implies wp\ (g\ t)\ P\ E\ s$ "
  shows " $wp\ (case\ a\ of\ MArray\ i\ m \Rightarrow f\ i\ m\ | MValue\ t \Rightarrow g\ t)\ P\ E\ s$ "
  <proof>
```

```
lemma wpstypes[wprule]:
  assumes " $\bigwedge i\ m. a = SArray\ i\ m \implies wp\ (f\ i\ m)\ P\ E\ s$ "
    and " $\bigwedge t\ t'. a = SMap\ t\ t' \implies wp\ (g\ t\ t')\ P\ E\ s$ "
    and " $\bigwedge t. a = SValue\ t \implies wp\ (h\ t)\ P\ E\ s$ "
  shows " $wp\ (case\ a\ of\ SArray\ i\ m \Rightarrow f\ i\ m\ | SMap\ t\ t' \Rightarrow g\ t\ t'\ | SValue\ t \Rightarrow h\ t)\ P\ E\ s$ "
  <proof>
```

```
lemma wptype[wprule]:
  assumes " $\bigwedge v. a = Value\ v \implies wp\ (f\ v)\ P\ E\ s$ "
    and " $\bigwedge m. a = Calldata\ m \implies wp\ (g\ m)\ P\ E\ s$ "
    and " $\bigwedge m. a = type.Memory\ m \implies wp\ (h\ m)\ P\ E\ s$ "
    and " $\bigwedge t. a = type.Storage\ t \implies wp\ (i\ t)\ P\ E\ s$ "
  shows " $wp\ (case\ a\ of\ Value\ v \Rightarrow f\ v\ | Calldata\ m \Rightarrow g\ m\ | type.Memory\ m \Rightarrow h\ m\ | type.Storage\ s \Rightarrow i\ s)\ P\ E\ s$ "
  <proof>
```

```
lemma wptypes[wprule]:
  assumes " $\bigwedge x. a = TSInt\ x \implies wp\ (f\ x)\ P\ E\ s$ "
    and " $\bigwedge x. a = TUInt\ x \implies wp\ (g\ x)\ P\ E\ s$ "
    and " $a = TBool \implies wp\ h\ P\ E\ s$ "
    and " $a = TAddr \implies wp\ i\ P\ E\ s$ "
  shows " $wp\ (case\ a\ of\ TSInt\ x \Rightarrow f\ x\ | TUInt\ x \Rightarrow g\ x\ | TBool \Rightarrow h\ | TAddr \Rightarrow i)\ P\ E\ s$ "
  <proof>
```

```
lemma wpltype[wprule]:
  assumes " $\bigwedge l. a = LStackloc\ l \implies wp\ (f\ l)\ P\ E\ s$ "
    and " $\bigwedge l. a = LMemloc\ l \implies wp\ (g\ l)\ P\ E\ s$ "
    and " $\bigwedge l. a = LStoreloc\ l \implies wp\ (h\ l)\ P\ E\ s$ "
  shows " $wp\ (case\ a\ of\ LStackloc\ l \Rightarrow f\ l\ | LMemloc\ l \Rightarrow g\ l\ | LStoreloc\ l \Rightarrow h\ l)\ P\ E\ s$ "
  <proof>
```

```
lemma wpdenvalue[wprule]:
  assumes " $\bigwedge l. a = Stackloc\ l \implies wp\ (f\ l)\ P\ E\ s$ "
    and " $\bigwedge l. a = Storeloc\ l \implies wp\ (g\ l)\ P\ E\ s$ "
```

```

  shows "wp (case a of Stackloc l ⇒ f l | Storeloc l ⇒ g l) P E s"
  ⟨proof⟩

```

7.2 Calculus (Weakest_Precondition)

7.2.1 Hoare Triples

```

context statement_with_gas
begin

```

```

type_synonym state_predicate = "accounts × stack × memoryT × (address ⇒ storageT) ⇒ bool"

```

```

definition validS :: "state_predicate ⇒ s ⇒ state_predicate ⇒ (ex ⇒ bool) ⇒ bool"
  ("{P}_S / _ / ({Q}_S, {E}_S)")

```

```

where

```

```

  "{P}_S s {Q}_S, {E}_S ≡
  ∀st. P (Accounts st, Stack st, Memory st, Storage st)
  → (∀ev cd. case stmt s ev cd st of
    Normal (_,st') ⇒ Q (Accounts st', Stack st', Memory st', Storage st')
    | Exception e ⇒ E e)"

```

```

definition wpS :: "s ⇒ (state ⇒ bool) ⇒ (ex ⇒ bool) ⇒ environment ⇒ calldataT ⇒ state ⇒ bool"
  where "wpS s P E ev cd st ≡ wp (λst. stmt s ev cd st) (λ_. P) E st"

```

```

lemma wpS_valid:

```

```

  assumes "∧ev cd (st::state). P (Accounts st, Stack st, Memory st, Storage st) ⇒ wpS s (λst. Q
  (Accounts st, Stack st, Memory st, Storage st)) E ev cd st"
  shows "{P}_S s {Q}_S, {E}_S"
  ⟨proof⟩

```

```

lemma valid_wpS:

```

```

  assumes "{P}_S s {Q}_S, {E}_S"
  shows "∧ev cd st. P (Accounts st, Stack st, Memory st, Storage st) ⇒ wpS s (λst. Q (Accounts st,
  Stack st, Memory st, Storage st)) E ev cd st"
  ⟨proof⟩

```

7.2.2 Skip

```

lemma wp_Skip:

```

```

  assumes "P (st(Gas := state.Gas st - costs SKIP ev cd st))"
  and "E Gas"
  shows "wpS SKIP P E ev cd st"
  ⟨proof⟩

```

7.2.3 Assign

```

lemma wp_Assign1:

```

```

  fixes ex ev cd st lv
  defines "ngas ≡ state.Gas st - costs (ASSIGN lv ex) ev cd st"
  assumes "∧rx ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]
  ⇒ is_KValue rx ∧ is_LStackloc lx"
  and "∧v t g l t' g' v'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KValue v, Value t), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((LStackloc l, Value t'), g');
  convert t t' v = Some v']]
  ⇒ P (st(Gas := g', Stack:=updateStore l (KValue v') (Stack st)))"
  and "∧e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e ⇒ E e"
  and "∧rx ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Exception e]]
  ⇒ E e"
  and "E Err"
  and "E Gas"

```

shows "wpS (ASSIGN lv ex) P E ev cd st"
 <proof>

lemma wp_Assign2:

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge$ rx ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
   $\implies$  is_KValue rx  $\wedge$  is_LStoreloc lx"
and " $\bigwedge$ v t g l t' g' v'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KValue v, Value t), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((LStoreloc l, type.Storage (STValue t')), g');
    convert t t' v = Some v']]
   $\implies$  P (st(Gas := g', Storage:= (Storage st) (Address ev := (fmupd l v' (Storage st (Address
ev))))))"
and " $\bigwedge$ e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e  $\implies$  E e"
and " $\bigwedge$ rx ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Exception e]]
   $\implies$  E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>
```

lemma wp_Assign3:

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge$ rx ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
   $\implies$  is_KValue rx  $\wedge$  is_LMemloc lx"
and " $\bigwedge$ v t g l t' g' v'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KValue v, Value t), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((LMemloc l, type.Memory (MTValue t')), g');
    convert t t' v = Some v']]
   $\implies$  P (st(Gas := g', Memory:=updateStore l (MValue v') (Memory st)))"
and " $\bigwedge$ e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e  $\implies$  E e"
and " $\bigwedge$ rx ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Exception e]]
   $\implies$  E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>
```

lemma wp_Assign4:

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge$ rx ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
   $\implies$  is_KCDptr rx  $\wedge$  is_LStackloc lx  $\wedge$  is_Memory ly"
and " $\bigwedge$ p x t g l t' g' p' m' sv'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KCDptr p, Calldata (MTArray x t)), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((LStackloc l, type.Memory t'), g');
    t' = MTArray x t;
    accessStore l (state.Stack st) = Some (KMemptr p');
    sv' = updateStore l (KMemptr (ShowLnat (Toploc (state.Memory st)))) (state.Stack st);
    cpm2m p (ShowLnat (Toploc (state.Memory st))) x t cd (snd (allocate(state.Memory st))) =
Some m']]
   $\implies$  P (st(Gas := g', Memory := m', Stack := sv'))"
```

```

and "∧e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e ⇒ E e"
and "∧rx ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Exception e]]
  ⇒ E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
⟨proof⟩

```

lemma wp_Assign5:

```

fixes ex ev cd st lv
defines "ngas ≡ state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes "∧rx ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
  ⇒ is_KCDptr rx ∧ is_LStackloc lx ∧ is_Storage ly"
and "∧p x t g l t' g' p' s'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KCDptr p, Calldata (MArray x t)), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((LStackloc l, type.Storage t'), g');
  accessStore l (Stack st) = Some (KStoptr p');
  cpm2s p p' x t cd (Storage st (Address ev)) = Some s']]
  ⇒ P (st(Gas := g', Storage:=(Storage st) (Address ev := s')))"
and "∧e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e ⇒ E e"
and "∧rx ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Exception e]]
  ⇒ E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
⟨proof⟩

```

lemma wp_Assign6:

```

fixes ex ev cd st lv
defines "ngas ≡ state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes "∧rx ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
  ⇒ is_KCDptr rx ∧ is_LStoreloc lx"
and "∧p x t g l t' g' s'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KCDptr p, Calldata (MArray x t)), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((LStoreloc l, t'), g');
  cpm2s p l x t cd (Storage st (Address ev)) = Some s']]
  ⇒ P (st(Gas := g', Storage:=(Storage st) (Address ev := s')))"
and "∧e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e ⇒ E e"
and "∧rx ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Exception e]]
  ⇒ E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
⟨proof⟩

```

lemma wp_Assign7:

```

fixes ex ev cd st lv
defines "ngas ≡ state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes "∧rx ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
  ⇒ is_KCDptr rx ∧ is_LMemloc lx"
and "∧p x t g l t' g' m m'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KCDptr p, Calldata (MArray x t)), g);

```

```

lexp lv ev cd (st(Gas := g)) g = Normal ((LMemloc l, type.Memory t'), g');
t' = MArray x t;
cpm2m p (ShowLnat (Toploc (state.Memory st))) x t cd (snd (allocate(state.Memory st))) =
Some m;
  m' = updateStore l (MPointer (ShowLnat (Toploc (state.Memory st)))) m]]
 $\implies$  P (st(Gas := g', Memory := m'))"
and " $\bigwedge e$ . expr ex ev cd (st(Gas := ngas)) ngas = Exception e  $\implies$  E e"
and " $\bigwedge rx$  ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Exception e]]
 $\implies$  E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>

```

lemma wp_Assign8:

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge rx$  ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
 $\implies$  is_KMemptr rx  $\wedge$  is_LStackloc lx  $\wedge$  is_Memory ly"
and " $\bigwedge p$  x t g l t' g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KMemptr p, type.Memory (MArray x t)), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((LStackloc l, type.Memory t'), g')]]
 $\implies$  P (st(Gas := g', Stack:=updateStore l (KMemptr p) (Stack st)))"
and " $\bigwedge e$ . expr ex ev cd (st(Gas := ngas)) ngas = Exception e  $\implies$  E e"
and " $\bigwedge rx$  ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Exception e]]
 $\implies$  E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>

```

lemma wp_Assign9:

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge rx$  ry g lx ly g'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
 $\implies$  is_KMemptr rx  $\wedge$  is_LStackloc lx  $\wedge$  is_Storage ly"
and " $\bigwedge p$  x t g l t' g' p' s'.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KMemptr p, type.Memory (MArray x t)), g);
  lexp lv ev cd (st(Gas := g)) g = Normal ((LStackloc l, type.Storage t'), g');
  accessStore l (Stack st) = Some (KStoptr p');
  cpm2s p p' x t (Memory st) (Storage st (Address ev)) = Some s']]
 $\implies$  P (st(Gas := g', Storage:=(Storage st) (Address ev := s')))"
and " $\bigwedge e$ . expr ex ev cd (st(Gas := ngas)) ngas = Exception e  $\implies$  E e"
and " $\bigwedge rx$  ry g e.
  [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(Gas := g)) g = Exception e]]
 $\implies$  E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>

```

lemma wp_Assign10:

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge rx$  ry g lx ly g'."

```

```

    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
  => is_KMemptr rx ^ is_LStoreloc lx"
and "^(p x t g l t' g' s'.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KMemptr p, type.Memory (MArray x t)), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((LStoreloc l, t'), g');
    cpm2s p l x t (Memory st) (Storage st (Address ev)) = Some s']]"
  => P (st(Gas := g', Storage:=Storage st) (Address ev := s')))"
and "^(e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e => E e)"
and "^(rx ry g e.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Exception e]]
  => E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>

```

lemma wp_Assign11:

```

fixes ex ev cd st lv
defines "ngas ≡ state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes "^(rx ry g lx ly g'.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
  => is_KMemptr rx ^ is_LMemloc lx"
and "^(p x t g l t' g'.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KMemptr p, type.Memory (MArray x t)), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((LMemloc l, t'), g')]]
  => P (st(Gas := g', Memory:=updateStore l (MPointer p) (Memory st)))"
and "^(e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e => E e)"
and "^(rx ry g e.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Exception e]]
  => E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>

```

lemma wp_Assign12:

```

fixes ex ev cd st lv
defines "ngas ≡ state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes "^(rx ry g lx ly g'.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
  => is_KStoptr rx ^ (∃rt. ry = type.Storage rt ^ is_STArray rt) ^ is_LStackloc lx ^
is_Memory ly"
and "^(p x t g l t' g' p' m sv'.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KStoptr p, type.Storage (STArray x t)), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((LStackloc l, type.Memory t'), g');
    cps2mTypeCompatible (STArray x t) t';
    accessStore l (state.Stack st) = Some (KMemptr p');
    sv' = updateStore l (KMemptr (ShowLnat (Toploc (state.Memory st)))) (state.Stack st);
    cps2m p (ShowLnat (Toploc (state.Memory st))) x t (state.Storage st (Address ev)) (snd
(allocate(state.Memory st))) = Some m]]
  => P (st(Gas := g', Memory := m, Stack := sv')))"
and "^(e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e => E e)"
and "^(rx ry g e.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Exception e]]
  => E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"

```

<proof>

lemma *wp_Assign13:*

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge$ rx ry g lx ly g'.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(|Gas := g|)) g = Normal ((lx,ly), g')]]
   $\implies$  is_KStoptr rx  $\wedge$  ( $\exists$ rt. ry = type.Storage rt  $\wedge$  is_STArray rt)  $\wedge$  is_LStackloc lx  $\wedge$ 
  is_Storage ly"
and " $\bigwedge$ p x t g l t' g'.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((KStoptr p, type.Storage (STArray x t)), g);
  lexp lv ev cd (st(|Gas := g|)) g = Normal ((LStackloc l, type.Storage t'), g')]]
   $\implies$  P (st(|Gas := g', Stack:=updateStore l (KStoptr p) (Stack st)|))"
and " $\bigwedge$ e. expr ex ev cd (st(|Gas := ngas|)) ngas = Exception e  $\implies$  E e"
and " $\bigwedge$ rx ry g e.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(|Gas := g|)) g = Exception e]]
   $\implies$  E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>

```

lemma *wp_Assign14:*

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge$ rx ry g lx ly g'.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(|Gas := g|)) g = Normal ((lx,ly), g')]]
   $\implies$  is_KStoptr rx  $\wedge$  is_LStoreloc lx"
and " $\bigwedge$ p x t g l t' g' s'.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((KStoptr p, type.Storage (STArray x t)), g);
  lexp lv ev cd (st(|Gas := g|)) g = Normal ((LStoreloc l, t'), g');
  copy p l x t (Storage st (Address ev)) = Some s']]
   $\implies$  P (st(|Gas := g', Storage:=(Storage st) (Address ev := s')|))"
and " $\bigwedge$ e. expr ex ev cd (st(|Gas := ngas|)) ngas = Exception e  $\implies$  E e"
and " $\bigwedge$ rx ry g e.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(|Gas := g|)) g = Exception e]]
   $\implies$  E e"
and "E Err"
and "E Gas"
shows "wpS (ASSIGN lv ex) P E ev cd st"
<proof>

```

lemma *wp_Assign15:*

```

fixes ex ev cd st lv
defines "ngas  $\equiv$  state.Gas st - costs (ASSIGN lv ex) ev cd st"
assumes " $\bigwedge$ rx ry g lx ly g'.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((rx,ry), g);
  lexp lv ev cd (st(|Gas := g|)) g = Normal ((lx,ly), g')]]
   $\implies$  is_KStoptr rx  $\wedge$  is_LMemloc lx"
and " $\bigwedge$ p x t g l t' g' m m'.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((KStoptr p, type.Storage (STArray x t)), g);
  lexp lv ev cd (st(|Gas := g|)) g = Normal ((LMemloc l, type.Memory t'), g');
  cps2mTypeCompatible (STArray x t) t';
  cps2m p (ShowLnat (Toploc (state.Memory st))) x t (state.Storage st (Address ev)) (snd
  (allocate(state.Memory st))) = Some m;
  m' = updateStore l (MPointer (ShowLnat (Toploc (state.Memory st)))) m]]
   $\implies$  P (st(|Gas := g', Memory := m'|))"
and " $\bigwedge$ e. expr ex ev cd (st(|Gas := ngas|)) ngas = Exception e  $\implies$  E e"
and " $\bigwedge$ rx ry g e.
  [[expr ex ev cd (st(|Gas := ngas|)) ngas = Normal ((rx,ry), g);

```

```

    lexp lv ev cd (st(Gas := g)) g = Exception e]]
  => E e"
  and "E Err"
  and "E Gas"
  shows "wpS (ASSIGN lv ex) P E ev cd st"
  <proof>

```

lemma wp_Assign16:

```

  fixes ex ev cd st lv
  defines "ngas ≡ state.Gas st - costs (ASSIGN lv ex) ev cd st"
  assumes "∧rx ry g lx ly g'.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((lx,ly), g')]]
    => is_KStoptr rx ∧ (∃rt. ry = type.Storage rt ∧ is_STMap rt) ∧ is_LStackloc lx"
  and "∧p t t' g l t'' g'.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((KStoptr p, type.Storage (STMap t t')), g);
    lexp lv ev cd (st(Gas := g)) g = Normal ((LStackloc l, t''), g')]]
    => P (st(Gas := g', Stack:=updateStore l (KStoptr p) (Stack st)))"
  and "∧e. expr ex ev cd (st(Gas := ngas)) ngas = Exception e => E e"
  and "∧rx ry g e.
    [[expr ex ev cd (st(Gas := ngas)) ngas = Normal ((rx,ry), g);
    lexp lv ev cd (st(Gas := g)) g = Exception e]]
    => E e"
  and "E Err"
  and "E Gas"
  shows "wpS (ASSIGN lv ex) P E ev cd st"
  <proof>

```

7.2.4 Composition

lemma wp_Comp:

```

  assumes "wpS s1 (wpS s2 P E ev cd) E ev cd (st(Gas := state.Gas st - costs (COMP s1 s2) ev cd st))"
  and "E Gas"
  shows "wpS (COMP s1 s2) P E ev cd st"
  <proof>

```

7.2.5 Conditional

lemma wp_ITE:

```

  fixes ex s1 s2 ev cd st
  defines "nGas ≡ state.Gas st - costs (ITE ex s1 s2) ev cd st"
  assumes "∧g. expr ex ev cd (st(Gas := nGas)) nGas = Normal ((KValue [True], Value TBool), g) =>
  wpS s1 P E ev cd (st(Gas := g))"
  and "∧g. expr ex ev cd (st(Gas := nGas)) nGas = Normal ((KValue [False], Value TBool), g) =>
  wpS s2 P E ev cd (st(Gas := g))"
  and "∧e. expr ex ev cd (st(Gas := nGas)) nGas = Exception e => E e"
  and "E Gas"
  and "E Err"
  shows "wpS (ITE ex s1 s2) P E ev cd st"
  <proof>

```

7.2.6 While Loop

lemma wp_While[rule_format]:

```

  fixes iv:"accounts × stack × memoryT × (address ⇒ storageT) ⇒ bool"
  and ex sm ev cd
  defines "nGas st ≡ state.Gas st - costs (WHILE ex sm) ev cd st"
  assumes "∧st g. [[iv (Accounts st, Stack st, Memory st, Storage st); expr ex ev cd (st(Gas := nGas
  st)) (nGas st) = Normal ((KValue [False], Value TBool), g)]] => P (st(Gas := g))"
  and "∧st g. [[iv (Accounts st, Stack st, Memory st, Storage st); expr ex ev cd (st(Gas := nGas
  st)) (nGas st) = Normal ((KValue [True], Value TBool), g)]] => wpS sm (λst. iv (Accounts st, Stack st,
  Memory st, Storage st)) E ev cd (st(Gas:=g))"
  and "∧st e. [[expr ex ev cd (st(Gas := nGas st)) (nGas st) = Exception e]] => E e"
  and "∧st e. stmt sm ev cd st = Exception e => E e"

```

```

    and "E Err"
    and "E Gas"
  shows "iv (Accounts st, Stack st, Memory st, Storage st)  $\longrightarrow$  wpS (WHILE ex sm) P E ev cd st"
  <proof>

```

7.2.7 Blocks

lemma wp_blockNone:

```

  fixes id0 tp stm ev cd st
  defines "nGas  $\equiv$  state.Gas st - costs (BLOCK (id0, tp, None) stm) ev cd st"
  assumes " $\bigwedge$ cd' mem' sck' e'. decl id0 tp None False cd (Memory st) (Storage st (Address ev))
    (cd, Memory st, Stack st, ev) = Some (cd', mem', sck', e')
     $\implies$  wpS stm P E e' cd' (st(|Gas := nGas, Stack := sck', Memory := mem'))"
  and "E Gas"
  and "E Err"
  shows "wpS (BLOCK (id0, tp, None) stm) P E ev cd st"
  <proof>

```

lemma wp_blockSome:

```

  fixes id0 tp ex' stm ev cd st
  defines "nGas  $\equiv$  state.Gas st - costs (BLOCK (id0, tp, Some ex') stm) ev cd st"
  assumes " $\bigwedge$ v t g' cd' mem' sck' e'.
    [[expr ex' ev cd (st(|Gas := nGas)) nGas = Normal ((v, t), g');
    decl id0 tp (Some (v,t)) False cd (Memory st) (Storage st (Address ev)) (cd, Memory st, Stack
    st, ev) = Some (cd', mem', sck', e')]]
     $\implies$  wpS stm P E e' cd' (st(|Gas := g', Stack := sck', Memory := mem'))"
  and " $\bigwedge$ e. expr ex' ev cd (st(|Gas := nGas)) nGas = Exception e  $\implies$  E e"
  and "E Gas"
  and "E Err"
  shows "wpS (BLOCK (id0, tp, Some ex') stm) P E ev cd st"
  <proof>

```

end

7.2.8 External method invocation

```

locale Calculus = statement_with_gas +
  fixes cname::Identifier
  and members:: "(Identifier, member) fmap"
  and const:: "(Identifier  $\times$  type) list  $\times$  s"
  and fb :: s
  assumes C1: "ep $$ cname = Some (members, const, fb)"
  begin

```

The rules for method invocations is provided in the context of four parameters:

- `cname::String.literal`: The name of the contract to be verified
- `members::(String.literal, member) fmap`: The member variables of the contract to be verified
- `const`: The constructor of the contract to be verified
- `fb`: The fallback method of the contract to be verified

In addition `C1` assigns members, constructor, and fallback method to the contract address.

An invariant is a predicate over two parameters:

- The private store of the contract
- The balance of the contract

```

type_synonym invariant = "storageT  $\Rightarrow$  int  $\Rightarrow$  bool"

```

7.2.9 Method invocations and transfer

definition Q_e

```

where "Qe ad iv st  $\equiv$ 
  ( $\forall$  mid fp f ev.
    members $$ mid = Some (Method (fp,True,f))  $\wedge$ 
    Address ev  $\neq$  ad
     $\rightarrow$  ( $\forall$  adex cd st' xe val g v t g' v' el cdl kl ml g'' acc.
      g''  $\leq$  state.Gas st  $\wedge$ 
      Type (acc ad) = Some (atype.Contract cname)  $\wedge$ 
      expr adex ev cd (st'(|Gas := state.Gas st' - costs (EXTERNAL adex mid xe val) ev cd st'|))
      (state.Gas st' - costs (EXTERNAL adex mid xe val) ev cd st') = Normal ((KValue ad, Value TAddr), g)  $\wedge$ 
      expr val ev cd (st'(|Gas := g|)) g = Normal ((KValue v, Value t), g')  $\wedge$ 
      convert t (TUInt b256) v = Some v'  $\wedge$ 
      load True fp xe (ffold (init members) (emptyEnv ad cname (Address ev) v') (fdom members))
      emptyTypedStore emptyStore emptyTypedStore ev cd (st'(|Gas := g'|)) g' = Normal ((el, cdl, kl, ml), g'')  $\wedge$ 
      transfer (Address ev) ad v' (Accounts (st'(|Gas := g'|))) = Some acc  $\wedge$ 
      iv (Storage st' ad) (ReadLint (Bal (acc ad)) - ReadLint v')
       $\rightarrow$  wpS f ( $\lambda$ st. iv (Storage st ad) (ReadLint (Bal (Accounts st ad)))) ( $\lambda$ e. e = Gas  $\vee$  e =
      Err) el cdl (st'(|Gas := g''), Accounts := acc, Stack := kl, Memory := ml)))")

```

definition Q_i

```

where "Qi ad pre post st  $\equiv$ 
  ( $\forall$  mid fp f ev.
    members $$ mid = Some (Method (fp,False,f))  $\wedge$ 
    Address ev = ad
     $\rightarrow$  ( $\forall$  cd st' i xe el cdl kl ml g.
      g  $\leq$  state.Gas st  $\wedge$ 
      load False fp xe (ffold (init members) (emptyEnv ad cname (Sender ev) (Svalue ev)) (fdom
      members)) emptyTypedStore emptyStore (Memory st') ev cd (st'(|Gas := state.Gas st' - costs (INVOKE i xe)
      ev cd st'|)) (state.Gas st' - costs (INVOKE i xe) ev cd st') = Normal ((el, cdl, kl, ml), g)  $\wedge$ 
      pre mid (ReadLint (Bal (Accounts st' ad)), Storage st' ad, el, cdl, kl, ml)
       $\rightarrow$  wpS f ( $\lambda$ st. post mid (ReadLint (Bal (Accounts st ad)), Storage st ad)) ( $\lambda$ e. e = Gas  $\vee$  e =
      Err) el cdl (st'(|Gas := g, Stack := kl, Memory := ml)))")

```

definition Q_{fi}

```

where "Qfi ad pref postf st  $\equiv$ 
  ( $\forall$  ev. Address ev = ad
     $\rightarrow$  ( $\forall$  ex cd st' adex cc v t g g' v' acc.
      g'  $\leq$  state.Gas st  $\wedge$ 
      expr adex ev cd (st'(|Gas := state.Gas st' - cc|)) (state.Gas st' - cc) = Normal ((KValue ad,
      Value TAddr), g)  $\wedge$ 
      expr ex ev cd (st'(|Gas := g|)) g = Normal ((KValue v, Value t), g')  $\wedge$ 
      convert t (TUInt b256) v = Some v'  $\wedge$ 
      transfer (Address ev) ad v' (Accounts st') = Some acc  $\wedge$ 
      pref (ReadLint (Bal (acc ad)), Storage st' ad)
       $\rightarrow$  wpS fb ( $\lambda$ st. postf (ReadLint (Bal (Accounts st ad)), Storage st ad)) ( $\lambda$ e. e = Gas  $\vee$  e =
      Err) (ffold (init members) (emptyEnv ad cname (Address ev) v') (fdom members)) emptyTypedStore (st'(|Gas
      := g', Accounts := acc, Stack := emptyStore, Memory := emptyTypedStore|)))")

```

definition Q_{fe}

```

where "Qfe ad iv st  $\equiv$ 
  ( $\forall$  ev. Address ev  $\neq$  ad
     $\rightarrow$  ( $\forall$  ex cd st' adex cc v t g g' v' acc.
      g'  $\leq$  state.Gas st  $\wedge$ 
      Type (acc ad) = Some (atype.Contract cname)  $\wedge$ 
      expr adex ev cd (st'(|Gas := state.Gas st' - cc|)) (state.Gas st' - cc) = Normal ((KValue ad,
      Value TAddr), g)  $\wedge$ 
      expr ex ev cd (st'(|Gas := g|)) g = Normal ((KValue v, Value t), g')  $\wedge$ 
      convert t (TUInt b256) v = Some v'  $\wedge$ 
      transfer (Address ev) ad v' (Accounts st') = Some acc  $\wedge$ 
      iv (Storage st' ad) (ReadLint (Bal (acc ad)) - ReadLint v')
       $\rightarrow$  wpS fb ( $\lambda$ st. iv (Storage st ad) (ReadLint (Bal (Accounts st ad)))) ( $\lambda$ e. e = Gas  $\vee$  e = Err)
      (ffold (init members) (emptyEnv ad cname (Address ev) v') (fdom members)) emptyTypedStore (st'(|Gas :=

```

$g', \text{Accounts} := \text{acc}, \text{Stack} := \text{emptyStore}, \text{Memory} := \text{emptyTypedStore})))))$ "

lemma *safeStore*[rule_format]:

```

fixes ad iv
defines "aux1 st  $\equiv \forall st'::\text{state}. \text{state.Gas } st' < \text{state.Gas } st \longrightarrow Qe \text{ ad iv } st'$ "
and "aux2 st  $\equiv \forall st'::\text{state}. \text{state.Gas } st' < \text{state.Gas } st \longrightarrow Qfe \text{ ad iv } st'$ "
shows " $\forall st'. \text{Address } ev \neq \text{ad} \wedge \text{Type } (\text{Accounts } st \text{ ad}) = \text{Some } (\text{atype.Contract } \text{cname}) \wedge \text{iv } (\text{Storage } st \text{ ad}) (\text{ReadL}_{int} (\text{Bal } (\text{Accounts } st \text{ ad}))) \wedge$ 
 $\text{stmt } f \text{ ev } cd \text{ st} = \text{Normal } ((), st') \wedge \text{aux1 } st \wedge \text{aux2 } st$ 
 $\longrightarrow \text{iv } (\text{Storage } st' \text{ ad}) (\text{ReadL}_{int} (\text{Bal } (\text{Accounts } st' \text{ ad})))$ "

```

<proof>

type_synonym precondition = "int \times storageT \times environment \times (mtypes,memoryvalue) typedstore \times stackvalue store \times (mtypes,memoryvalue) typedstore \Rightarrow bool"

type_synonym postcondition = "int \times storageT \Rightarrow bool"

The following lemma can be used to verify (recursive) internal or external method calls and transfers executed from ****inside**** ($\text{Address } ev = \text{ad}$). In particular the lemma requires the contract to be annotated as follows:

- Pre/Postconditions for internal methods
- Invariants for external methods

The lemma then requires us to verify the following:

- Postconditions from preconditions for internal method bodies.
- Invariants hold for external method bodies.

To this end it allows us to assume the following:

- Preconditions imply postconditions for internal method calls.
- Invariants hold for external method calls for other Contracts external methods.

definition *Pe*

```

where "Pe ad iv st  $\equiv$ 
 $(\forall ev \text{ ad}' \text{ i xe val } cd.$ 
 $\text{Address } ev = \text{ad} \wedge$ 
 $(\forall adv \text{ c g v t g}' \text{ v}'.$ 
 $\text{expr } ad' \text{ ev } cd \text{ (st} \langle \text{Gas} := \text{state.Gas } st - \text{costs } (\text{EXTERNAL } ad' \text{ i xe val}) \text{ ev } cd \text{ st} \rangle) \text{ (state.Gas } st - \text{costs } (\text{EXTERNAL } ad' \text{ i xe val}) \text{ ev } cd \text{ st}) = \text{Normal } ((\text{KValue } adv, \text{Value } TAddr), g) \wedge$ 
 $adv \neq \text{ad} \wedge$ 
 $\text{Type } (\text{Accounts } st \text{ adv}) = \text{Some } (\text{atype.Contract } c) \wedge$ 
 $c \in | \text{fmdom } ep \wedge$ 
 $\text{expr } val \text{ ev } cd \text{ (st} \langle \text{Gas} := g \rangle) g = \text{Normal } ((\text{KValue } v, \text{Value } t), g') \wedge$ 
 $\text{convert } t \text{ (TUInt } b256) v = \text{Some } v'$ 
 $\longrightarrow \text{iv } (\text{Storage } st \text{ ad}) (\text{ReadL}_{int} (\text{Bal } (\text{Accounts } st \text{ ad})) - \text{ReadL}_{int} v')$ 
 $\longrightarrow \text{wpS } (\text{EXTERNAL } ad' \text{ i xe val}) (\lambda st. \text{iv } (\text{Storage } st \text{ ad}) (\text{ReadL}_{int} (\text{Bal } (\text{Accounts } st \text{ ad})))) (\lambda e. e = \text{Gas} \vee e = \text{Err}) \text{ ev } cd \text{ st})$ "

```

definition *Pi*

```

where "Pi ad pre post st  $\equiv$ 
 $(\forall ev \text{ i xe } cd.$ 
 $\text{Address } ev = \text{ad} \wedge$ 
 $\text{Contract } ev = \text{cname} \wedge$ 
 $(\forall fp \text{ e}_l \text{ cd}_l \text{ k}_l \text{ m}_l \text{ g}.$ 
 $\text{load } \text{False } fp \text{ xe } (\text{ffold } (\text{init members}) (\text{emptyEnv } ad \text{ (Contract } ev) (\text{Sender } ev) (\text{Svalue } ev))$ 
 $(\text{fmdom members})) \text{emptyTypedStore } \text{emptyStore } (\text{Memory } st) \text{ ev } cd \text{ (st} \langle \text{Gas} := \text{state.Gas } st - \text{costs } (\text{INVOKE } i \text{ xe}) \text{ ev } cd \text{ st} \rangle) \text{ (state.Gas } st - \text{costs } (\text{INVOKE } i \text{ xe}) \text{ ev } cd \text{ st}) = \text{Normal } ((e_l, cd_l, k_l, m_l), g)$ 
 $\longrightarrow \text{pre } i \text{ (ReadL}_{int} (\text{Bal } (\text{Accounts } st \text{ ad})), \text{Storage } st \text{ ad}, e_l, cd_l, k_l, m_l)$ 
 $\longrightarrow \text{wpS } (\text{INVOKE } i \text{ xe}) (\lambda st. \text{post } i \text{ (ReadL}_{int} (\text{Bal } (\text{Accounts } st \text{ ad})), \text{Storage } st \text{ ad})) (\lambda e. e = \text{Gas} \vee e = \text{Err}) \text{ ev } cd \text{ st})$ "

```

definition *Pfi*

where "Pfi ad pref postf st \equiv

7 Verification Support

```

(∀ ev ex ad' cd.
  Address ev = ad ∧
  (∀ adv g.
    expr ad' ev cd (st(|Gas := state.Gas st - costs (TRANSFER ad' ex) ev cd st|)) (state.Gas st -
costs (TRANSFER ad' ex) ev cd st) = Normal ((KValue adv, Value TAddr), g)
    → adv = ad) ∧
  (∀ g v t g'.
    expr ad' ev cd (st(|Gas := state.Gas st - costs (TRANSFER ad' ex) ev cd st|)) (state.Gas st -
costs (TRANSFER ad' ex) ev cd st) = Normal ((KValue ad, Value TAddr), g) ∧
    expr ex ev cd (st(|Gas := g|)) g = Normal ((KValue v, Value t), g')
    → pref (ReadLint (Bal (Accounts st ad)), Storage st ad)
    → wpS (TRANSFER ad' ex) (λst. postf (ReadLint (Bal (Accounts st ad))), Storage st ad)) (λe. e = Gas
∨ e = Err) ev cd st)"

```

definition Pfe

```

where "Pfe ad iv st ≡
  (∀ ev ex ad' cd.
    Address ev = ad ∧
    (∀ adv g.
      expr ad' ev cd (st(|Gas := state.Gas st - costs (TRANSFER ad' ex) ev cd st|)) (state.Gas st -
costs (TRANSFER ad' ex) ev cd st) = Normal ((KValue adv, Value TAddr), g)
      → adv ≠ ad) ∧
    (∀ adv g v t g' v'.
      expr ad' ev cd (st(|Gas := state.Gas st - costs (TRANSFER ad' ex) ev cd st|)) (state.Gas st -
costs (TRANSFER ad' ex) ev cd st) = Normal ((KValue adv, Value TAddr), g) ∧
      adv ≠ ad ∧
      expr ex ev cd (st(|Gas := g|)) g = Normal ((KValue v, Value t), g') ∧
      convert t (TUInt b256) v = Some v'
      → iv (Storage st ad) (ReadLint (Bal (Accounts st ad)) - ReadLint v')
      → wpS (TRANSFER ad' ex) (λst. iv (Storage st ad) (ReadLint (Bal (Accounts st ad)))) (λe. e = Gas
∨ e = Err) ev cd st)"

```

lemma wp_external_invoke_transfer:

```

fixes pre::"Identifier ⇒ precondition"
and post::"Identifier ⇒ postcondition"
and pref::"postcondition"
and postf::"postcondition"
and iv::"invariant"
assumes asm: "∧st::state.
  [∀st'::state. state.Gas st' ≤ state.Gas st ∧ Type (Accounts st' ad) = Some (atype.Contract cname)
    → Pe ad iv st' ∧ Pi ad pre post st' ∧ Pfi ad pref postf st' ∧ Pfe ad iv st']
  ⇒ Qe ad iv st ∧ Qi ad pre post st ∧ Qfi ad pref postf st ∧ Qfe ad iv st"
shows "Type (Accounts st ad) = Some (atype.Contract cname) → Pe ad iv st ∧ Pi ad pre post st ∧
Pfi ad pref postf st ∧ Pfe ad iv st"
⟨proof⟩

```

Refined versions of `wp_external_invoke_transfer`.

lemma wp_transfer_ext[rule_format]:

```

assumes "Type (Accounts st ad) = Some (atype.Contract cname)"
and "∧st::state. [∀st'::state. state.Gas st' ≤ state.Gas st ∧ Type (Accounts st' ad) = Some
(atype.Contract cname) → Pe ad iv st' ∧ Pi ad pre post st' ∧ Pfi ad pref postf st' ∧ Pfe ad iv st']
  ⇒ Qe ad iv st ∧ Qi ad pre post st ∧ Qfi ad pref postf st ∧ Qfe ad iv st"
shows "(∀ ev ex ad' cd.
  Address ev = ad ∧
  (∀ adv g.
    expr ad' ev cd (st(|Gas := state.Gas st - costs (TRANSFER ad' ex) ev cd st|)) (state.Gas st -
costs (TRANSFER ad' ex) ev cd st) = Normal ((KValue adv, Value TAddr), g)
    → adv ≠ ad) ∧
  (∀ adv g v t g' v'.
    expr ad' ev cd (st(|Gas := state.Gas st - costs (TRANSFER ad' ex) ev cd st|)) (state.Gas st -
costs (TRANSFER ad' ex) ev cd st) = Normal ((KValue adv, Value TAddr), g) ∧
    adv ≠ ad ∧
    expr ex ev cd (st(|Gas := g|)) g = Normal ((KValue v, Value t), g') ∧
    convert t (TUInt b256) v = Some v'
  )
)"

```

$\rightarrow iv$ (Storage st ad) (ReadL_{int} (Bal (Accounts st ad)) - ReadL_{int} v'))
 $\rightarrow wpS$ (TRANSFER ad' ex) ($\lambda st. iv$ (Storage st ad) (ReadL_{int} (Bal (Accounts st ad)))) ($\lambda e. e = Gas \vee e = Err$) ev cd st)"
 <proof>

lemma wp_external[rule_format]:

assumes "Type (Accounts st ad) = Some (atype.Contract cname)"
 and " $\bigwedge st::state. \llbracket \forall st':state. state.Gas\ st' \leq state.Gas\ st \wedge Type\ (Accounts\ st'\ ad) = Some\ (atype.Contract\ cname) \rrbracket \rightarrow Pe\ ad\ iv\ st' \wedge Pi\ ad\ pre\ post\ st' \wedge Pfi\ ad\ pref\ postf\ st' \wedge Pfe\ ad\ iv\ st' \rrbracket$
 $\implies Qe\ ad\ iv\ st \wedge Qi\ ad\ pre\ post\ st \wedge Qfi\ ad\ pref\ postf\ st \wedge Qfe\ ad\ iv\ st$ "
 shows " $(\forall ev\ ad'\ i\ xe\ val\ cd. Address\ ev = ad \wedge (\forall adv\ c\ g\ v\ t\ g'\ v'. expr\ ad'\ ev\ cd\ (st\ (Gas := state.Gas\ st - costs\ (EXTERNAL\ ad'\ i\ xe\ val)\ ev\ cd\ st))\ (state.Gas\ st - costs\ (EXTERNAL\ ad'\ i\ xe\ val)\ ev\ cd\ st) = Normal\ ((KValue\ adv,\ Value\ TAddr),\ g) \wedge adv \neq ad \wedge Type\ (Accounts\ st\ adv) = Some\ (atype.Contract\ c) \wedge c \in |fmdom\ ep \wedge expr\ val\ ev\ cd\ (st\ (Gas := g))\ g = Normal\ ((KValue\ v,\ Value\ t),\ g') \wedge convert\ t\ (TUInt\ b256)\ v = Some\ v')$
 $\rightarrow iv$ (Storage st ad) (ReadL_{int} (Bal (Accounts st ad)) - ReadL_{int} v'))
 $\rightarrow wpS$ (EXTERNAL ad' i xe val) ($\lambda st. iv$ (Storage st ad) (ReadL_{int} (Bal (Accounts st ad)))) ($\lambda e. e = Gas \vee e = Err$) ev cd st)"
 <proof>

lemma wp_invoke[rule_format]:

assumes "Type (Accounts st ad) = Some (atype.Contract cname)"
 and " $\bigwedge st::state. \llbracket \forall st':state. state.Gas\ st' \leq state.Gas\ st \wedge Type\ (Accounts\ st'\ ad) = Some\ (atype.Contract\ cname) \rrbracket \rightarrow Pe\ ad\ iv\ st' \wedge Pi\ ad\ pre\ post\ st' \wedge Pfi\ ad\ pref\ postf\ st' \wedge Pfe\ ad\ iv\ st' \rrbracket$
 $\implies Qe\ ad\ iv\ st \wedge Qi\ ad\ pre\ post\ st \wedge Qfi\ ad\ pref\ postf\ st \wedge Qfe\ ad\ iv\ st$ "
 shows " $(\forall ev\ i\ xe\ cd. Address\ ev = ad \wedge Contract\ ev = cname \wedge (\forall fp\ e_i\ cd_i\ k_i\ m_i\ g. load\ False\ fp\ xe\ (ffold\ (init\ members)\ (emptyEnv\ ad\ (Contract\ ev)\ (Sender\ ev)\ (Svalue\ ev))\ (fmdom\ members))\ emptyTypedStore\ emptyStore\ (Memory\ st)\ ev\ cd\ (st\ (Gas := state.Gas\ st - costs\ (INVOKE\ i\ xe)\ ev\ cd\ st))\ (state.Gas\ st - costs\ (INVOKE\ i\ xe)\ ev\ cd\ st) = Normal\ ((e_i,\ cd_i,\ k_i,\ m_i),\ g)$
 $\rightarrow pre\ i$ (ReadL_{int} (Bal (Accounts st ad)), Storage st ad, e_i, cd_i, k_i, m_i))
 $\rightarrow wpS$ (INVOKE i xe) ($\lambda st. post\ i$ (ReadL_{int} (Bal (Accounts st ad)), Storage st ad)) ($\lambda e. e = Gas \vee e = Err$) ev cd st)"
 <proof>

lemma wp_transfer_int[rule_format]:

assumes "Type (Accounts st ad) = Some (atype.Contract cname)"
 and " $\bigwedge st::state. \llbracket \forall st':state. state.Gas\ st' \leq state.Gas\ st \wedge Type\ (Accounts\ st'\ ad) = Some\ (atype.Contract\ cname) \rrbracket \rightarrow Pe\ ad\ iv\ st' \wedge Pi\ ad\ pre\ post\ st' \wedge Pfi\ ad\ pref\ postf\ st' \wedge Pfe\ ad\ iv\ st' \rrbracket$
 $\implies Qe\ ad\ iv\ st \wedge Qi\ ad\ pre\ post\ st \wedge Qfi\ ad\ pref\ postf\ st \wedge Qfe\ ad\ iv\ st$ "
 shows " $(\forall ev\ ex\ ad'\ cd. Address\ ev = ad \wedge (\forall adv\ g. expr\ ad'\ ev\ cd\ (st\ (Gas := state.Gas\ st - costs\ (TRANSFER\ ad'\ ex)\ ev\ cd\ st))\ (state.Gas\ st - costs\ (TRANSFER\ ad'\ ex)\ ev\ cd\ st) = Normal\ ((KValue\ adv,\ Value\ TAddr),\ g)$
 $\rightarrow adv = ad) \wedge (\forall g\ v\ t\ g'. expr\ ad'\ ev\ cd\ (st\ (Gas := state.Gas\ st - costs\ (TRANSFER\ ad'\ ex)\ ev\ cd\ st))\ (state.Gas\ st - costs\ (TRANSFER\ ad'\ ex)\ ev\ cd\ st) = Normal\ ((KValue\ ad,\ Value\ TAddr),\ g) \wedge expr\ ex\ ev\ cd\ (st\ (Gas := g))\ g = Normal\ ((KValue\ v,\ Value\ t),\ g')$
 $\rightarrow pref$ (ReadL_{int} (Bal (Accounts st ad)), Storage st ad))
 $\rightarrow wpS$ (TRANSFER ad' ex) ($\lambda st. postf$ (ReadL_{int} (Bal (Accounts st ad)), Storage st ad)) ($\lambda e. e = Gas \vee e = Err$) ev cd st)"
 <proof>

definition constructor :: " $((String.literal, String.literal) fmap \Rightarrow int \Rightarrow bool) \Rightarrow bool$ "

where "constructor iv $\equiv (\forall acc\ g''\ m_l\ k_l\ cd_l\ e_l\ g'\ t\ v\ v'\ xe\ i\ cd\ val\ st\ ev\ adv\ st0.$

```

st0 = st(Gas := state.Gas st - costs (NEW i xe val) ev cd st) ∧
adv = hash_version (Address ev) (ShowLnat (Contracts (Accounts st0 (Address ev)))) ∧
Type (Accounts st0 adv) = None ∧
expr val ev cd st0 (state.Gas st0) = Normal ((KValue v, Value t), g') ∧
convert t (TUInt b256) v = Some v' ∧
load True (fst const) xe (ffold (init members) (emptyEnv adv cname (Address ev) v') (fmdom members))
emptyTypedStore emptyStore emptyTypedStore ev cd (st0(Gas := g', Accounts := (Accounts st0)(adv := (Bal
= ShowLint 0, Type = Some (atype.Contract cname), Contracts = 0)), Storage:=(Storage st0)(adv := {$$})))
g' = Normal ((ei, cdi, ki, mi), g'') ∧
transfer (Address ev) adv v' ((Accounts st0)(adv := (Bal = ShowLint 0, Type = Some (atype.Contract
cname), Contracts = 0))) = Some acc
→ wpS (snd const) (λst. iv (Storage st adv) [Bal (Accounts st adv)]) (λe. e = Gas ∨ e = Err) ei
cdi
(st0(Gas := g'', Storage:=(Storage st0)(adv := {$$}), Accounts := acc, Stack:=ki, Memory:=mi)))"

```

lemma invariant_rec:

```

fixes iv ad
assumes "∀ ad (st::state). Qe ad iv st"
and "∀ ad (st::state). Qfe ad iv st"
and "constructor iv"
and "Address ev ≠ ad"
and "Type (Accounts st ad) = Some (atype.Contract cname) → iv (Storage st ad) (ReadLint (Bal
(Accounts st ad)))"
shows "∀ (st::state). stmt f ev cd st = Normal (((), st') ∧ Type (Accounts st' ad) = Some
(atype.Contract cname)
→ iv (Storage st' ad) (ReadLint (Bal (Accounts st' ad))))"
⟨proof⟩

```

theorem invariant:

```

fixes iv ad
assumes "∀ ad (st::state). Qe ad iv st"
and "∀ ad (st::state). Qfe ad iv st"
and "constructor iv"
and "∀ ad. Type (Accounts st ad) = Some (atype.Contract cname) → iv (Storage st ad) (ReadLint
(Bal (Accounts st ad)))"
shows "∀ (st::state) ad. stmt f ev cd st = Normal (((), st') ∧ Type (Accounts st' ad) = Some
(atype.Contract cname) ∧ Address ev ≠ ad
→ iv (Storage st' ad) (ReadLint (Bal (Accounts st' ad))))"
⟨proof⟩
end

```

context Calculus

begin

```

named_theorems external
named_theorems internal

```

7.3 Verification Condition Generator (Weakest_Precondition)

To use the verification condition generator first invoke the following rule on the original Hoare triple:

```

method vcg_valid =
  rule wpS_valid,
  erule conjE,
  simp

method external uses cases =
  unfold Qe_def,
  elims,
  (erule cases; simp)

method fallback uses cases =
  unfold Qfe_def,
  elims,

```

```

rule cases,
simp

method constructor uses cases =
  unfold constructor_def,
  elims,
  rule cases,
  simp

```

Then apply the correct rules from the following set of rules.

7.3.1 Skip

```

method vcg_skip =
  rule wp_Skip; (solve simp)?

```

7.3.2 Assign

```

method vcg_assign uses wp_rule expr_rule lexp_rule =
  rule wp_rule,
  simp add: expr_rule lexp_rule,
  simp

```

7.3.3 Composition

```

method vcg_comp =
  rule wp_Comp; simp

```

7.3.4 Blocks

```

method vcg_block_some =
  rule wp_blockSome; simp
end

```

```

locale VCG = Calculus +
  fixes iv::invariant
  and pref::"postcondition"
  and postf::"postcondition"
  and pre::"Identifier ⇒ precondition"
  and post::"Identifier ⇒ postcondition"
begin

```

7.3.5 Transfer

The following rule can be used to verify an invariant for a transfer statement. It requires four term parameters:

- $iv::(\text{String.literal}, \text{String.literal}) \text{ fmap} \Rightarrow \text{int} \Rightarrow \text{bool}$: Invariant
- $pref::\text{int} \times (\text{String.literal}, \text{String.literal}) \text{ fmap} \Rightarrow \text{bool}$: Precondition for fallback method called internally
- $postf::\text{int} \times (\text{String.literal}, \text{String.literal}) \text{ fmap} \Rightarrow \text{bool}$: Postcondition for fallback method called internally
- $pre::\text{String.literal} \Rightarrow \text{int} \times (\text{String.literal}, \text{String.literal}) \text{ fmap} \times \text{environment} \times (\text{mtypes}, \text{memoryvalue}) \text{ typedstore} \times \text{stackvalue store} \times (\text{mtypes}, \text{memoryvalue}) \text{ typedstore} \Rightarrow \text{bool}$: Preconditions for internal methods
- $post::\text{String.literal} \Rightarrow \text{int} \times (\text{String.literal}, \text{String.literal}) \text{ fmap} \Rightarrow \text{bool}$: Postconditions for internal methods

In addition it requires 8 facts:

- $fallback_int$: verifies **postcondition** for body of fallback method invoked **internally**.
- $fallback_ext$: verifies **invariant** for body of fallback method invoked **externally**.

7 Verification Support

- `cases_ext`: performs case distinction over **external** methods of `atype.Contract ad`.
- `cases_int`: performs case distinction over **internal** methods of `atype.Contract ad`.
- `cases_fb`: performs case distinction over **fallback** method of `atype.Contract ad`.
- `different`: shows that Address of environment is different from `ad`.
- `invariant`: shows that invariant holds **before** execution of transfer statement.

Finally it requires two lists of facts as parameters:

- `external`: verify that the invariant is preserved by the body of external methods.
- `internal`: verify that the postcondition holds after executing the body of internal methods.

```
method vcg_prep =
  (rule allI)+,
  rule impI,
  (erule conjE)+
```

```
method vcg_body uses fallback_int fallback_ext cases_ext cases_int cases_fb =
  (rule conjI)?,
  match conclusion in
  "Qe _ _ _" =>
    <unfold Qe_def,
      vcg_prep,
      erule cases_ext;
      (simp,vcg_prep,
        rule external;
        solve <assumption / simp>>>
  | "Qi _ _ _ _" =>
    <unfold Qi_def,
      vcg_prep,
      erule cases_int;
      (simp,vcg_prep,
        rule internal;
        solve <assumption / simp>>>
  | "Qfi _ _ _ _" =>
    <unfold Qfi_def,
      rule allI,
      rule impI,
      rule cases_fb;
      (simp,vcg_prep,
        rule fallback_int;
        solve <assumption / simp>>>
  | "Qfe _ _ _" =>
    <unfold Qfe_def,
      rule allI,
      rule impI,
      rule cases_fb;
      (simp,vcg_prep,
        rule fallback_ext;
        solve <assumption / simp>>>
```

```
method decl_load_rec for ad::address and e::environment uses eq decl load empty init =
  match premises in
  d: "decl _ _ _ _ _ (, , , e') = Some (, , , e)" for e'::environment =>
    <decl_load_rec ad e' eq:trans_sym[OF eq decl[OF d]] decl:decl load:load empty:empty init:init>
  | l: "load _ _ _ (ffold (init members) (emptyEnv ad cname (Address e') v) (fmdom members)) _ _ _ _ _
    = Normal ((e, , , ,))" for e'::environment and v =>
    <rule
      trans[
        OF eq
        trans[
```

```

    OF load[OF l]
    trans[
      OF init[of (unchecked) members "emptyEnv ad cname (Address e') v" "fmdom members"]
      empty[of (unchecked) ad cname "Address e'" v]]]]>

method sameaddr for ad::address =
  match conclusion in
    "Address e = ad" for e::environment =>
      <decl_load_rec ad e eq:refl[of "Address e"] decl:decl_env[THEN conjunct1]
load:msel_ssel_expr_load_rexp_gas(4)[THEN conjunct2, THEN conjunct1] init:ffold_init_ad
empty:emptyEnv_address>

lemma eq_neq_eq_imp_neq:
  "x = a ==> b ≠ y ==> a = b ==> x ≠ y" <proof>

method sender for ad::address =
  match conclusion in
    "adv ≠ ad" for adv::address =>
      <match premises in
        a: "Address e' ≠ ad" and e: "expr SENDER e _ _ = Normal ((KValue adv, Value TAddr), _)" for
e::environment and e':environment =>
          <rule local.eq_neq_eq_imp_neq[OF expr_sender[OF e] a],
            decl_load_rec ad e eq:refl[of "Sender e"] decl:decl_env[THEN conjunct2, THEN
conjunct1] load:msel_ssel_expr_load_rexp_gas(4)[THEN conjunct2, THEN conjunct2, THEN conjunct1]
init:ffold_init_sender empty:emptyEnv_sender>>

method vcg_init for ad::address uses invariant =
  elims,
  sameaddr ad,
  sender ad,
  (rule invariant; assumption)

method vcg_transfer_ext for ad::address
  uses fallback_int fallback_ext cases_ext cases_int cases_fb invariant =
  rule wp_transfer_ext[where pref = pref and postf = postf and pre = pre and post = post and
iv=iv],
  solve simp,
  (vcg_body fallback_int:fallback_int fallback_ext:fallback_ext cases_ext:cases_ext cases_int:cases_int
cases_fb:cases_fb)+,
  vcg_init ad invariant:invariant

end

end

```


8 Applications

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

8.1 Reentrancy (Reentrancy)

In the following we use our calculus to verify a contract implementing a simple token. The contract is defined by definition **bank** 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 **BANK**) 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 implicit 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.

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 without calling "deposit". In such a case the difference would change. In particular it would grow. However, we were able to verify that the difference does never shrink which is what we actually want to ensure.

```
theory Reentrancy
  imports Weakest_Precondition Solidity_Evaluator
begin
```

8.1.1 Example of Re-entrancy

```
definition "example_env ≡
  loadProc (STR ''Attacker'')
    ([,
     ([, SKIP),
     ITE (LESS (BALANCE THIS) (UINT b256 125))
        (EXTERNAL (ADDRESS (STR ''BankAddress'')) (STR ''withdraw'') [] (UINT b256 0))
        SKIP)
  (loadProc (STR ''Bank'')
    ([ (STR ''balance'', Var (STMap TAddr (STValue (TUInt b256))))),
     (STR ''deposit'', Method ([, True,
      ASSIGN
        (Ref (STR ''balance'') [SENDER])
        (PLUS (LVAL (Ref (STR ''balance'') [SENDER])) VALUE))),
     (STR ''withdraw'', Method ([, True,
      ITE (LESS (UINT b256 0) (LVAL (Ref (STR ''balance'') [SENDER])))
      (COMP
        (TRANSFER SENDER (LVAL (Ref (STR ''balance'') [SENDER]))))
```

```

      (ASSIGN (Ref (STR ''balance'') [SENDER]) (UINT b256 0)))
    SKIP)),
  ([, SKIP),
  SKIP)
  fmempty)"

```

global_interpretation reentrancy: statement_with_gas costs_ex example_env costs_min

```

defines stmt = "reentrancy.stmt"
  and lexp = reentrancy.lexp
  and expr = reentrancy.expr
  and ssel = reentrancy.ssel
  and rexp = reentrancy.rexp
  and msel = reentrancy.msel
  and load = reentrancy.load
  and eval = reentrancy.eval
<proof>

```

lemma "eval 1000

```

  (COMP
    (EXTERNAL (ADDRESS (STR ''BankAddress'')) (STR ''deposit'') [] (UINT b256 10))
    (EXTERNAL (ADDRESS (STR ''BankAddress'')) (STR ''withdraw'') [] (UINT b256 0)))
  (STR ''AttackerAddress'')
  (STR ''Attacker'')
  (STR ''')
  (STR ''0'')
  [(STR ''BankAddress'', STR ''100'', atype.Contract (STR ''Bank''), 0), (STR
''AttackerAddress'', STR ''100'', atype.Contract (STR ''Attacker''), 0)]
  []
  = STR ''BankAddress: balance==70 - Bank(balance[AttackerAddress]==0) AttackerAddress:
balance==130 - Attacker()'')
<proof>

```

8.1.2 Definition of contract

abbreviation myrexp::l

where "myrexp \equiv Ref (STR ''balance'') [SENDER]"

abbreviation mylval::e

where "mylval \equiv LVAL myrexp"

abbreviation assign::s

where "assign \equiv ASSIGN myrexp (UINT b256 0)"

abbreviation transfer::s

where "transfer \equiv TRANSFER SENDER (LVAL (Id (STR ''bal'')))"

abbreviation comp::s

where "comp \equiv COMP assign transfer"

abbreviation keep::s

where "keep \equiv BLOCK (STR ''bal'', Value (TUInt b256), Some mylval) comp"

abbreviation deposit::s

where "deposit \equiv ASSIGN myrexp (PLUS mylval VALUE)"

abbreviation "banklist \equiv [

```

  (STR ''balance'', Var (STMap TAddr (STValue (TUInt b256)))),
  (STR ''deposit'', Method ([, True, deposit)),
  (STR ''withdraw'', Method ([, True, keep]))]"

```

definition bank::"(Identifier, member) fmap"

where "bank \equiv fmap_of_list banklist"

8.1.3 Verification

```

locale Reentrancy = Calculus +
  assumes r0: "cname = STR ''Bank''"
  and r1: "members = bank"
  and r2: "fb = SKIP"
  and r3: "const = ([, SKIP)"
begin

```

Method lemmas

These lemmas are required by `vcg_external`.

```

lemma mwithdraw[simp]:
  "members $$ STR ''withdraw'' = Some (Method ([, True, keep))"
  <proof>

```

```

lemma mdeposit[simp]:
  "members $$ STR ''deposit'' = Some (Method ([, True, deposit))"
  <proof>

```

Variable lemma

```

lemma balance[simp]:
  "members $$ (STR ''balance'') = Some (Var (STMap TAddr (STValue (TUInt b256))))"
  <proof>

```

```

lemma balAcc:
  "members $$ (STR ''bal'') = None"
  <proof>

```

Case lemmas

These lemmas are required by `vcg_transfer`.

```

lemma cases_ext:
  assumes "members $$ mid = Some (Method (fp,True,f))"
  and "fp = []  $\implies$  f = deposit  $\implies$  P"
  and "fp = []  $\implies$  f = keep  $\implies$  P"
  shows "P"
  <proof>

```

```

lemma cases_int:
  assumes "members $$ mid = Some (Method (fp,False,f))"
  shows "P"
  <proof>

```

```

lemma cases_fb:
  assumes "fb = SKIP  $\implies$  P"
  shows "P"
  <proof>

```

```

lemma cases_cons:
  assumes "fst const = []  $\implies$  snd const = SKIP  $\implies$  P"
  shows "P"
  <proof>

```

Definition of Invariant

abbreviation $SUMM\ s \equiv \sum (ad,x) | fmlookup\ s\ (ad + (STR\ ''.'' + STR\ ''balance'')) = Some\ x.\ ReadL_{int}\ x$

abbreviation $POS\ s \equiv \forall ad\ x.\ fmlookup\ s\ (ad + (STR\ ''.'' + STR\ ''balance'')) = Some\ x \longrightarrow ReadL_{int}\ x \geq 0$

definition $iv\ s\ a \equiv a \geq SUMM\ s \wedge POS\ s$

lemma weaken:

```

  assumes "iv (Storage st ad) (ReadLint (Bal (acc ad)) - ReadLint v)"
    and "ReadLint v ≥ 0"
  shows "iv (Storage st ad) (ReadLint (Bal (acc ad)))"
⟨proof⟩

```

Additional lemmas

lemma expr_0:

```

  assumes "expr (UINT b256 0) ev0 cd0 st0 g0 = Normal ((rv, rt), g''a)"
  shows "is_KValue rv" and "is_Value rt"
⟨proof⟩

```

lemma load_empty_par:

```

  assumes "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st(|Gas := g1|)) g1 = Normal ((el, cdl, kl, ml), g1'"
  shows "load True [] [] (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st(|Gas := g1|)) g1 = Normal ((el, cdl, kl, ml), g1'"
⟨proof⟩

```

lemma lexp_myrexp_decl:

```

  assumes "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st(|Gas := g1|)) g1 = Normal ((el, cdl, kl, ml), g1'"
  and "decl STR ''bal'' (Value (TUInt b256)) (Some (lv, lt)) False cdl ml (Storage st0 (Address el))
(cdl, ml, kl, el) = Some (cd', mem', sck', e'"
  and "lexp myrexp e' cd' (st0(|Accounts := acc, Stack := sck', Memory := mem', Gas := g'a|)) g'a =
Normal ((rv,rt), g''a)"
  shows "rv = LStoreloc (Address env + (STR ''.''' + STR ''balance''))" and "rt = type.Storage (STValue
(TUInt b256))"
⟨proof⟩

```

lemma lexp_myrexp_decl2:

```

  assumes "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st(|Gas := g1|)) g1 = Normal ((el, cdl, kl, ml), g1'"
  and "decl STR ''bal'' (Value (TUInt b256)) (Some (lv, lt)) False cdl ml (Storage st0 (Address el))
(cdl, ml, kl, el) = Some (cd', mem', sck', e'"
  and "lexp myrexp e' cd' (st0(|Accounts := acc, Stack := sck', Memory := mem', Gas := g'a|)) g'a =
Normal ((rv,rt), g''a)"
  shows "is_LStoreloc rv" and "(∃ lt. rt = type.Storage lt ∧ is_STValue lt)"
⟨proof⟩

```

lemma expr_bal:

```

  assumes "expr (LVAL (l.Id STR ''bal'')) e' cd' (st(|Accounts := acc, Stack := sck', Memory := mem',
Gas := g''a, Storage := (Storage st) (Address e' := fmupd l v' s'), Gas := g''|)) g'' = Normal ((KValue
lv, Value t), g'')"
  and "(sck', e') = astack_dup STR ''bal'' (Value (TUInt b256)) (KValue (accessStorage (TUInt b256)
(Address env + (STR ''.''' + STR ''balance'')) s')) (kl, el)"
  and "Denvalue el $$ STR ''bal'' = None"
  shows "([accessStorage (TUInt b256) (Address env + (STR ''.''' + STR ''balance'')) s']::int) =
ReadLint lv" (is ?G1) and "t = TUInt b256"
⟨proof⟩

```

lemma lexp_myrexp:

```

  assumes "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st(|Gas := g1|)) g1 = Normal ((el, cdl, kl, ml), g1'"
  and "lexp myrexp el cdl (st'(|Gas := g2|)) g2 = Normal ((rv,rt), g2'"
  shows "rv = LStoreloc (Address env + (STR ''.''' + STR ''balance''))" and "rt = type.Storage (STValue
(TUInt b256))"
⟨proof⟩

```

lemma expr_balance:

```

  assumes "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st(|Gas := g1|)) g1 = Normal ((el, cdl, kl, ml), g1'"
  and "expr (LVAL (Ref (STR ''balance'')) [SENDER])) el cdl (st(|Accounts := acc, Stack := kl, Memory

```

```

:= ml, Gas := g2) g2 = Normal ((va, ta), g'a)"
  shows "va= KValue (accessStorage (TUInt b256) (Address env + (STR ''.''' + STR ''balance'')))
(Storage st ad)"
  and "ta = Value (TUInt b256)"
⟨proof⟩

```

lemma `expr_plus`:

```

assumes "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st(|Gas := g3|)) g3 = Normal ((el, cdl, kl, ml), g3'"
  and "expr (PLUS a0 b0) ev0 cd0 st0 g0 = Normal ((xs, t'0), g'0)"
  shows "is_KValue xs" and "is_Value t'0"
⟨proof⟩

```

lemma `lexp_myrexp2`:

```

assumes "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st(|Gas := g1|)) g1 = Normal ((el, cdl, kl, ml), g1'"
  and "lexp myrexp el cdl (st'(|Gas := g2|)) g2 = Normal ((rv,rt), g2'"
  shows "is_LStoreloc rv" and "∃x. rt=type.Storage x ∧ is_STValue x"
⟨proof⟩

```

lemma `summ_eq_sum`:

```

"SUMM s' = (∑ (ad,x)|fmlookup s' (ad + (STR ''.''' + STR ''balance''))) = Some x ∧ ad ≠ adr. ReadLint
x)
  + ReadLint (accessStorage (TUInt b256) (adr + (STR ''.''' + STR ''balance''))) s'"
⟨proof⟩

```

lemma `sum_eq_update`:

```

assumes s''_def: "s'' = fmupd (adr + (STR ''.''' + STR ''balance''))) v' s'"
  shows "(∑ (ad,x)|fmlookup s'' (ad + (STR ''.''' + STR ''balance''))) = Some x ∧ ad ≠ adr. ReadLint
x)
  = (∑ (ad,x)|fmlookup s' (ad + (STR ''.''' + STR ''balance''))) = Some x ∧ ad ≠ adr. ReadLint
x)"
⟨proof⟩

```

lemma `adapt_deposit`:

```

assumes "Address env ≠ ad"
  and "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore emptyStore emptyTypedStore env cd (st0(|Gas := g3|)) g3 = Normal ((el, cdl, kl, ml), g3'"
  and "Accounts.transfer (Address env) ad v a = Some acc"
  and "iv (Storage st0 ad) (ReadLint (Bal (acc ad)) - ReadLint v)"
  and "lexp myrexp el cdl (st0(|Gas := g''', Accounts := acc, Stack := kl, Memory := ml, Gas := g'))
g' = Normal ((LStoreloc l, type.Storage (STValue t')), g''a)"
  and "expr (PLUS mylval VALUE) el cdl (st0(|Gas := g''', Accounts := acc, Stack := kl, Memory := ml,
Gas := g|)) g = Normal ((KValue va, Value ta), g'"
  and "Valuetypes.convert ta t' va = Some v'"
  shows "(ad = Address el → iv (Storage st0 (Address el)(1 $$$ = v')) [Bal (acc (Address el))] ∧
(ad ≠ Address el → iv (Storage st0 ad) (ReadLint (Bal (acc ad))))"
⟨proof⟩

```

lemma `adapt_withdraw`:

```

fixes st acc sck' mem' g''a e' l v' xe
  defines "st' ≡ st(|Accounts := acc, Stack := sck', Memory := mem', Gas := g''a, Storage := (Storage
st) (Address e' := (Storage st (Address e')) (1 $$$ = v')))"
  assumes "iv (Storage st ad) (ReadLint (Bal (acc ad)) - ReadLint v)"
  and "load True [] xe (ffold (init members) (emptyEnv ad cname (Address env) v) (fmdom members))
emptyTypedStore
  emptyStore emptyTypedStore env cd (st(|Gas := g'|)) g' = Normal ((el, cdl, kl, ml), g'"
  and "decl STR ''bal'' (Value (TUInt b256)) (Some (va, ta)) False cdl ml (Storage st (Address el))
(cdl, ml, kl, el) =
  Some (cd', mem', sck', e'"
  and "expr (UINT b256 0) e' cd' (st(|Accounts := acc, Stack := sck', Memory := mem', Gas := ga|)) ga
=
  Normal ((KValue vb, Value tb), g'b)"
  and "Valuetypes.convert tb t' vb = Some v'"

```

```

and "lexp myrexp e' cd' (st(|Accounts := acc, Stack := sck', Memory := mem', Gas := g'b|)) g'b =
  Normal ((LStoreloc l, type.Storage (STValue t')), g'a)"
and "expr mylval e_l cd_l (st(|Accounts := acc, Stack := k_l, Memory := m_l,
  Gas := g'' - costs keep e_l cd_l (st(|Gas := g'', Accounts := acc, Stack := k_l, Memory := m_l|)))
  (g'' - costs keep e_l cd_l (st(|Gas := g'', Accounts := acc, Stack := k_l, Memory := m_l|))) =
Normal ((va, ta), g'a)"
and "Accounts.transfer (Address env) ad v (Accounts st) = Some acc"
and Bal: "expr (LVAL (l.Id STR ''bal'')) e' cd' (st(|Gas := g''b|)) g''b = Normal ((KValue lv,
Value t), g'')"
and con: "Valuetypes.convert t (TUInt b256) lv = Some lv"
shows "iv (Storage st' ad) (ReadL_int (Bal (Accounts st' ad)) - (ReadL_int lv))"
<proof>

```

```

lemma expr_ex:
  assumes "local.expr x ev cd st g = Exception e"
  shows "e = ex.Gas ∨ e = Err"
<proof>

```

```

lemma lexp_ex:
  assumes "local.lexp x ev cd st g = Exception e"
  shows "e = ex.Gas ∨ e = Err"
<proof>

```

```

lemma wp_deposit[external]:
  assumes "Address ev ≠ ad"
  and "load True [] xe (ffold (init members) (emptyEnv ad cname (Address ev) v) (fndom members))
emptyTypedStore emptyStore emptyTypedStore ev cd (st(|Gas := g'|)) g' = Normal ((e_l, cd_l, k_l, m_l), g'')"
  and "Accounts.transfer (Address ev) ad v (Accounts st) = Some acc"
  and "iv (Storage st ad) (ReadL_int (Bal (acc ad)) - ReadL_int v)"
  shows "wpS deposit
  (λst. (iv (Storage st ad) (ReadL_int (Bal (Accounts st ad))))) (λe. e = Gas ∨ e = Err) e_l cd_l
  (st(|Gas := g'', Accounts := acc, Stack := k_l, Memory := m_l|))"
<proof>

```

```

lemma wptransfer:
  fixes st0 acc sck' mem' g'a e' l v'
  defines "st' ≡ st0(|Accounts := acc, Stack := sck', Memory := mem', Gas := g'a,
  Storage := (Storage st0)(Address e' := Storage st0 (Address e')(l $$$ := v'))|)"
  assumes "Pfe ad iv st'"
  and "Address ev ≠ ad"
  and "load True [] xe (ffold (init members) (emptyEnv ad cname (Address ev) gv') (fndom members))
emptyTypedStore
  emptyStore emptyTypedStore ev cd (st0(|Gas := g'|)) g' =
  Normal ((e_l, cd_l, k_l, m_l), g'')"
  and "Accounts.transfer (Address ev) ad gv' (Accounts st0) = Some acc"
  and "iv (Storage st0 ad) (ReadL_int (Bal (acc ad)) - ReadL_int gv')"
  and "decl STR ''bal'' (Value (TUInt b256)) (Some (v, t)) False cd_l m_l (Storage st0 (Address e_l))
  (cd_l, m_l, k_l, e_l) = Some (cd', mem', sck', e')"
  and "Valuetypes.convert ta t' va = Some v'"
  and "lexp myrexp e' cd' (st0(|Accounts := acc, Stack := sck', Memory := mem', Gas := g'a|)) g'a =
  Normal ((LStoreloc l, type.Storage (STValue t')), g'a)"
  and "expr mylval e_l cd_l (st0(|Accounts := acc, Stack := k_l, Memory := m_l,
  Gas := g'' - costs (BLOCK (STR ''bal'', Value (TUInt b256), Some mylval)
  (COMP (ASSIGN myrexp (UINT b256 0)) Reentrancy.transfer)) e_l cd_l (st0(|Gas := g'', Accounts :=
acc, Stack := k_l, Memory := m_l|)))
  (g'' - costs (BLOCK (STR ''bal'', Value (TUInt b256), Some mylval) (COMP (ASSIGN myrexp (UINT
b256 0)) Reentrancy.transfer)) e_l
  cd_l (st0(|Gas := g'', Accounts := acc, Stack := k_l, Memory := m_l|))) =
  Normal ((v, t), g'')"
  and "expr (UINT b256 0) e' cd' (st0(|Accounts := acc, Stack := sck', Memory := mem', Gas := ga|))
ga = Normal ((KValue va, Value ta), g'a)"
  shows "wpS Reentrancy.transfer (λst. iv (Storage st ad) (ReadL_int (Bal (Accounts st ad))))) (λe. e =
Gas ∨ e = Err) e' cd' st'"
<proof>

```

```

lemma wp_withdraw[external]:
  assumes "\st'. state.Gas st' ≤ state.Gas st ∧ Type (Accounts st' ad) = Some (atype.Contract cname)
  ⇒ Pe ad iv st' ∧ Pi ad (λ_ . True) (λ_ . True) st' ∧ Pfi ad (λ_ . True) (λ_ . True) st' ∧ Pfe ad iv
  st'"
  and "Address ev ≠ ad"
  and "g'' ≤ state.Gas st"
  and "Type (acc ad) = Some (atype.Contract cname)"
  and "load True [] xe (ffold (init members) (emptyEnv ad cname (Address ev) v') (fmdom members))
  emptyTypedStore emptyStore emptyTypedStore ev cd (st0(Gas := g')) g' = Normal ((el, cdl, kl,
  ml), g'')"
  and "Accounts.transfer (Address ev) ad v' (Accounts st0) = Some acc"
  and "iv (Storage st0 ad) (ReadLint (Bal (acc ad)) - ReadLint v'"
  shows "wpS keep (λst. iv (Storage st ad) (ReadLint (Bal (Accounts st ad)))) (λe. e = Gas ∨ e =
  Err) el cdl
  (st0(Gas := g'', Accounts := acc, Stack := kl, Memory := ml))"
  ⟨proof⟩

```

```

lemma wp_fallback:
  assumes "Accounts.transfer (Address ev) ad v (Accounts st) = Some acc"
  and "iv (Storage st ad) (ReadLint (Bal (acc ad)) - ReadLint v)"
  shows "wpS SKIP (λst. iv (Storage st ad) (ReadLint (Bal (Accounts st ad)))) (λe. e = Gas ∨ e =
  Err)
  (ffold (init members) (emptyEnv ad cname (Address ev) v) (fmdom members)) emptyTypedStore
  (st(Gas := g, Accounts := acc, Stack := emptyStore, Memory := emptyTypedStore))"
  ⟨proof⟩

```

```

lemma wp_construct:
  fixes ev st
  defines "adv ≡ hash_version (Address ev) [Contracts (Accounts st (Address ev))]"
  assumes "Accounts.transfer (Address ev) adv v ((Accounts st) (adv := (Bal = ShowLint 0, Type = Some
  (atype.Contract i), Contracts = 0))) = Some acc"
  shows "iv fmempty [Bal (acc adv)]"
  ⟨proof⟩

```

```

lemma wp_true:
  assumes "E Gas"
  and "E Err"
  shows "wpS f (λst. True) E e cd st"
  ⟨proof⟩

```

Final results

```

interpretation vcg:VCG costse ep costs cname members const fb iv "λ_ . True" "λ_ . True" "λ_ . True"
"λ_ . True"
⟨proof⟩

```

```

lemma safe_external: "Qe ad iv (st::state)"
  ⟨proof⟩

```

```

lemma safe_fallback: "Qfe ad iv st"
  ⟨proof⟩

```

```

lemma safe_constructor: "constructor iv"
  ⟨proof⟩

```

```

theorem safe:
  assumes "∀ad. Type (Accounts st ad) = Some (atype.Contract cname) → iv (Storage st ad) (ReadLint
  (Bal (Accounts st ad)))"
  shows "∀(st::state) ad. stmt f ev cd st = Normal (((), st') ∧ Type (Accounts st' ad) = Some
  (atype.Contract cname) ∧ Address ev ≠ ad
  → iv (Storage st' ad) (ReadLint (Bal (Accounts st' ad))))"
  ⟨proof⟩

```

end

end

8.2 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 xs) = Ref i (map eupdate xs)"
```

```
| "eupdate (e.INT b v) =
```

```
  (if v ≥ 0
```

```
    then e.INT b (-(2bits.to_nat b - 1) + (v+2bits.to_nat b - 1) mod (2bits.to_nat b))
```

```
    else e.INT b (2bits.to_nat b - 1 - (-v+2bits.to_nat b - 1) - 1 mod (2bits.to_nat b - 1))"
```

```
| "eupdate (UINT b v) = UINT b (v mod (2bits.to_nat b))"
```

```
| "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 ⇒
```

```
      (case (eupdate ex2) of
```

```
        e.INT b2 v2 ⇒
```

```
          let v=v1+v2 in
```

```
            if v ≥ 0
```

```
              then e.INT (max b1 b2) (-(2(max (bits.to_nat b1) (bits.to_nat b2))-1) + (v+2(max (bits.to_nat b1) (bits.to_nat b2))-1) mod (2(max (bits.to_nat b1) (bits.to_nat b2))))
```

```
              else e.INT (max b1 b2) (2(max (bits.to_nat b1) (bits.to_nat b2))-1 - (-v+2(max (bits.to_nat b1) (bits.to_nat b2))-1) - 1 mod (2(max (bits.to_nat b1) (bits.to_nat b2)) - 1))
```

```
        | UINT b2 v2 ⇒
```

```
          if b2 < b1
```

```
            then let v=v1+v2 in
```

```
              if v ≥ 0
```

```
                then e.INT b1 (-(2(bits.to_nat b1)-1) + (v+2(bits.to_nat b1)-1) mod (2(bits.to_nat b1)))
```

```
                else e.INT b1 (2(bits.to_nat b1)-1 - (-v+2(bits.to_nat b1)-1) - 1 mod (2(bits.to_nat b1) - 1))
```

```
              else PLUS (e.INT b1 v1) (UINT b2 v2)
```

```
        | _ ⇒ PLUS (e.INT b1 v1) (eupdate ex2))
```

```
    | UINT b1 v1 ⇒
```

```
      (case (eupdate ex2) of
```

```
        UINT b2 v2 ⇒ UINT (max b1 b2) ((v1 + v2) mod (2(max (bits.to_nat b1) (bits.to_nat b2))))
```

```
        | e.INT b2 v2 ⇒
```

```
          if b1 < b2
```

```
            then let v=v1+v2 in
```

```
              if v ≥ 0
```

```
                then e.INT b2 (-(2(bits.to_nat b2)-1) + (v+2(bits.to_nat b2)-1) mod (2(bits.to_nat b2)))
```

```
                else e.INT b2 (2(bits.to_nat b2)-1 - (-v+2(bits.to_nat b2)-1) - 1 mod (2(bits.to_nat b2) - 1))
```

```
              else PLUS (UINT b1 v1) (e.INT b2 v2)
```

```

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

```

```

        else EQUAL (UINT b1 v1) (e.INT b2 v2)
      | _ ⇒ EQUAL (UINT b1 v1) (eupdate ex2))
    | _ ⇒ EQUAL (eupdate ex1) (eupdate ex2))"
| "eupdate (LESS ex1 ex2) =
  (case (eupdate ex1) of
    e.INT b1 v1 ⇒
      (case (eupdate ex2) of
        e.INT b2 v2 ⇒
          if v1 < v2
            then TRUE
            else FALSE
        | UINT b2 v2 ⇒
          if 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))
    | UINT b1 v1 ⇒
      (case (eupdate ex2) of
        UINT b2 v2 ⇒
          if v1 < v2
            then TRUE
            else FALSE
        | e.INT b2 v2 ⇒
          if 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))
    | _ ⇒ 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) = ECALL e i xs"
| "eupdate CONTRACTS = CONTRACTS"

```

lemma "eupdate (UINT b8 250) = UINT b8 250" *<proof>*

```

lemma "eupdate (UINT b8 500) = UINT b8 244" <proof>
lemma "eupdate (e.INT b8 (-100)) = e.INT b8 (- 100)" <proof>
lemma "eupdate (e.INT b8 (-150)) = e.INT b8 106" <proof>
lemma "eupdate (PLUS (UINT b8 100) (UINT b8 100)) = UINT b8 200" <proof>
lemma "eupdate (PLUS (UINT b8 257) (UINT b16 100)) = UINT b16 101" <proof>
lemma "eupdate (PLUS (e.INT b8 100) (UINT b8 250)) = PLUS (e.INT b8 100) (UINT b8 250)" <proof>
lemma "eupdate (PLUS (e.INT b8 250) (UINT b8 500)) = PLUS (e.INT b8 (- 6)) (UINT b8 244)" <proof>
lemma "eupdate (PLUS (e.INT b16 250) (UINT b8 500)) = e.INT b16 494" <proof>
lemma "eupdate (EQUAL (UINT b16 250) (UINT b8 250)) = TRUE" <proof>
lemma "eupdate (EQUAL (e.INT b16 100) (UINT b8 100)) = TRUE" <proof>
lemma "eupdate (EQUAL (e.INT b8 100) (UINT b8 100)) = EQUAL (e.INT b8 100) (UINT b8 100)" <proof>

lemma update_bounds_int:
  assumes "eupdate ex = (e.INT b v)"
  shows "(v < 2^((bits.to_nat b)-1)) ∧ v ≥ -(2^((bits.to_nat b)-1))"
<proof>

lemma update_bounds_uint:
  assumes "eupdate ex = UINT b v"
  shows "v < 2^(bits.to_nat b) ∧ v ≥ 0"
<proof>

lemma no_gas:
  assumes "¬ g > costs_ex ex env cd st"
  shows "expr ex env cd st g = Exception Gas"
<proof>

lemma lift_eq:
  assumes "expr e1 env cd st g = expr e1' env cd st g"
  and "∧g' rv. expr e1 env cd st g = Normal (rv, g') ⇒ expr e2 env cd st g' = expr e2' env cd st g'"
  shows "lift expr f e1 e2 env cd st g = lift expr f e1' e2' env cd st g"
<proof>

lemma ssel_eq_ssel:
  "(∧i g. i ∈ set ix ⇒ expr i env cd st g = expr (f i) env cd st g)
  ⇒ ssel tp loc ix env cd st g = ssel tp loc (map f ix) env cd st g"
<proof>

lemma msel_eq_msel:
  "(∧i g. i ∈ set ix ⇒ expr i env cd st g = expr (f i) env cd st g) ⇒
  msel c tp loc ix env cd st g = msel c tp loc (map f ix) env cd st g"
<proof>

lemma ref_eq:
  assumes "∧e g. e ∈ set ex ⇒ expr e env cd st g = expr (f e) env cd st g"
  shows "rexp (Ref i ex) env cd st g = rexp (Ref i (map f ex)) env cd st g"
<proof>

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

theorem update_correctness:
  "∧g. expr ex env cd st g = expr (eupdate ex) env cd st g"
  "∧g. rexp lv env cd st g = rexp (lupdate lv) env cd st g"
<proof>

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>.