Isabelle/Solidity

A deep Embedding of Solidity in Isabelle/HOL

Diego Marmsoler and Achim D. Brucker and Billy Thornton

March 19, 2025

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

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

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

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

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

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

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

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

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

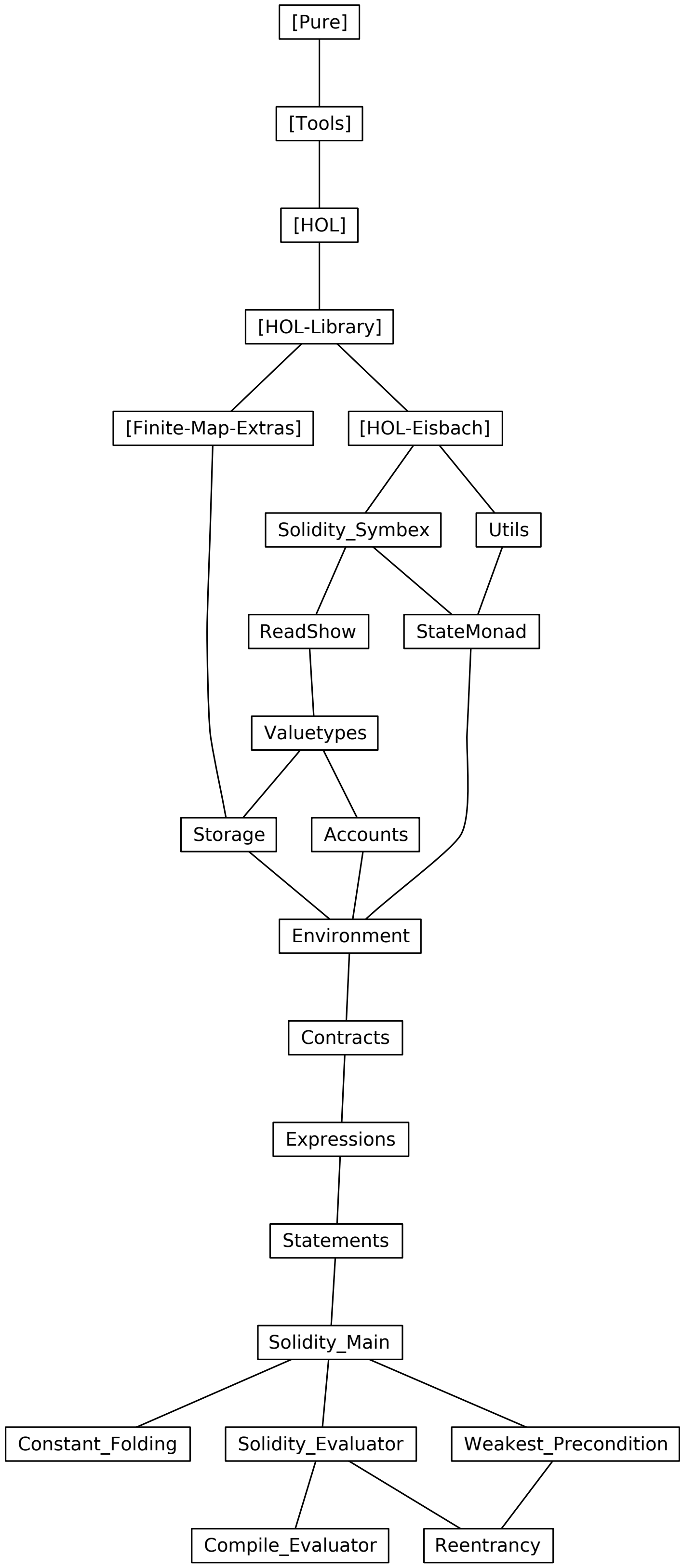

Figure 1.1: The Dependency Graph of the Isabelle Theories.

2 Preliminaries

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

2.1 Converting Types to Strings and Back Again (ReadShow)

```
theory ReadShow
  imports
    Solidity_Symbex
begin
```

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

Bool

```
definition
 ‹Read_bool s = (if s = ''True'' then True else False)›
definition
 ‹Show_bool b = (if b then ''True'' else ''False'')›
definition
 ‹STR_is_bool s = (Show_bool (Read_bool s) = s)›

declare Read_bool_def [solidity_symbex]
        Show_bool_def [solidity_symbex]

lemma Show_Read_bool_id: ‹STR_is_bool s ⟹ (Show_bool (Read_bool s) = s)›
  ⟨proof⟩

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

lemma Read_Show_bool_id: ‹Read_bool (Show_bool b) = b›
  ⟨proof⟩

definition ReadL_bool::‹String.literal ⇒ bool› (‹⌊_⌋›) where
 ‹ReadL_bool s = (if s = STR ''True'' then True else False)›
definition ShowL_bool:: ‹bool ⇒ String.literal› (‹⌈_⌉›) where
 ‹ShowL_bool b = (if b then STR ''True'' else STR ''False'')›
definition
 ‹strL_is_bool' s = (ShowL_bool (ReadL_bool s) = s)›

declare ReadL_bool_def [solidity_symbex]
        ShowL_bool_def [solidity_symbex]


lemma Show_Read_bool'_id: ‹strL_is_bool' s ⟹ (ShowL_bool (ReadL_bool s) = s)›
  ⟨proof⟩

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

lemma Read_Show_bool'_id[simp]: ‹ReadL_bool (ShowL_bool b) = b›
  ⟨proof⟩
```

```
lemma true_neq_false[simp]: "ShowL_bool True ≠ ShowL_bool False"
  ⟨proof⟩
```

Natural Numbers

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

declare nat_of_digit_def [solidity_symbex]

definition is_digit :: ‹char ⇒ 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 if c = CHR ''-'' then True
    else False)›

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

declare digit_of_nat_def [solidity_symbex]

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

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

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

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

declare nat_implode_def [solidity_symbex]

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

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

declare nat_explode_def [solidity_symbex]

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

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

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

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

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

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

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

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

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

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


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

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

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

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

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

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

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

definition
Show$_{nat}$::"nat ⇒ string" **where**
‹Show$_{nat}$ n = map digit_of_nat (nat_explode n)›

declare Read$_{nat}$_def [solidity_symbex]
Show$_{nat}$_def [solidity_symbex]

definition
‹STR_is_nat s = (Show$_{nat}$ (Read$_{nat}$ s) = s)›

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

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

lemma Show_nat_not_empty: ‹(Show$_{nat}$ n) ≠ []›
⟨proof⟩

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

lemma Show_nat_not_neg'': ‹hd (Show$_{nat}$ n) ≠ (CHR ''-'')›
⟨proof⟩

lemma Show_Read_nat_id: ‹STR_is_nat s ⟹ (Show$_{nat}$ (Read$_{nat}$ s) = s)›
⟨proof⟩

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

lemma Read_Show_nat_id: ‹Read$_{nat}$(Show$_{nat}$ n) = n›
⟨proof⟩

definition
ReadL$_{nat}$ :: ‹String.literal ⇒ nat› (‹⌈_⌉›) **where**
‹ReadL$_{nat}$ = Read$_{nat}$ ∘ String.explode›

definition
ShowL$_{nat}$::‹nat ⇒ String.literal› (‹⌊_⌋›)**where**
‹ShowL$_{nat}$ = String.implode ∘ Show$_{nat}$›

declare ReadL$_{nat}$_def [solidity_symbex]
ShowL$_{nat}$_def [solidity_symbex]

definition
‹strL_is_nat' s = (ShowL$_{nat}$ (ReadL$_{nat}$ s) = s)›

value ‹⌈STR ''10''⌉::nat›
value ‹ReadL$_{nat}$ (STR ''10'')›
value ‹⌊10::nat⌋›
value ‹ShowL$_{nat}$ 10›
value ‹ReadL$_{nat}$ (ShowL$_{nat}$ (10)) = 10›
value ‹ShowL$_{nat}$ (ReadL$_{nat}$ (STR ''10'')) = STR ''10''›

lemma Show_Read_nat'_id: ‹strL_is_nat' s ⟹ (ShowL$_{nat}$ (ReadL$_{nat}$ s) = s)›
⟨proof⟩

lemma digits_are_ascii:
‹c ∈ {CHR ''0'', CHR ''1'', CHR ''2'', CHR ''3'', CHR ''4'',
CHR ''5'', CHR ''6'', CHR ''7'', CHR ''8'', CHR ''9''}
⟹ String.ascii_of c = c›
⟨proof⟩

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

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

Integer

definition
Read$_{int}$ :: ‹string ⇒ int› **where**
‹Read$_{int}$ x = (if hd x = (CHR ''-'') then -(int (Read$_{nat}$ (tl x))) else int (Read$_{nat}$ x))›

definition
Show$_{int}$::‹int ⇒ string› **where**
‹Show$_{int}$ i = (if i < 0 then (CHR ''-'')#(Show$_{nat}$ (nat (-i)))
else Show$_{nat}$ (nat i))›

definition
‹STR_is_int s = (Show$_{int}$ (Read$_{int}$ s) = s)›

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

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

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

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

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

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

definition
$ReadL_{int}$ :: ‹String.literal ⇒ int› (‹⌈_⌉›) **where**
‹$ReadL_{int}$ = $Read_{int}$ ∘ String.explode›

definition
$ShowL_{int}$::‹int ⇒ String.literal› (‹⌊_⌋›) **where**
‹$ShowL_{int}$ =String.implode ∘ $Show_{int}$›

definition
‹strL_is_int' s = ($ShowL_{int}$ ($ReadL_{int}$ s) = s)›

declare $ReadL_{int}$_def [solidity_symbex]
$ShowL_{int}$_def [solidity_symbex]

value ‹$ReadL_{int}$ ($ShowL_{int}$ 10) = 10›
value ‹$ReadL_{int}$ ($ShowL_{int}$ (-10)) = -10›

value ‹$ShowL_{int}$ ($ReadL_{int}$ (STR ''10'')) = STR ''10''›
value ‹$ShowL_{int}$ ($ReadL_{int}$ (STR ''-10'')) = STR ''-10''›

lemma Show_ReadL_id: ‹strL_is_int' s ⟹ ($ShowL_{int}$ ($ReadL_{int}$ s) = s)›
⟨*proof*⟩

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

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

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

end
theory Utils
imports
Main
"HOL-Eisbach.Eisbach"
begin

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

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

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

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

2.2 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⟩
```

2.2.1 Fundamental Definitions

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

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

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

adhoc_overloading Monad_Syntax.bind ⇌ bind

lemma throw_left[simp]: "throw x ≫= y = throw x" ⟨proof⟩
```

2.2.2 The Monad Laws

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

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

return is absorbed on the right of a *(≫=)*

```
lemma bind_return [simp]: "(m ≫= return) = m"
⟨proof⟩
```

(≫=) is associative

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

2.2.3 Basic Conguruence Rules

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


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

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

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

⟨proof⟩

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

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

```
lemma bind_case_bool_cong [fundef_cong]:
  assumes "x = x'" and "x = True ⟹ f s = f' s" and "x = False ⟹ g s = g' s"
  shows "(case x of True ⇒ f | False ⇒ g) s = (case x' of True ⇒ f' | False ⇒ g') s"
  ⟨proof⟩
```

2.2.4 Other functions

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

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

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

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

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

Modify the current state using the function passed in.

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

```
fun
  assert :: "'e ⇒ ('s ⇒ bool) ⇒ (unit, 'e, 's) state_monad" where
 "assert x t = (λs. if (t s) then return () s else throw x s)"
```

```
fun
  option :: "'e ⇒ ('s ⇒ 'a option) ⇒ ('a, 'e, 's) state_monad" where
 "option x f = (λs. (case f s of
    Some y ⇒ return y s
  | None ⇒ throw x s))"
```

2.2.5 Some basic examples

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

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

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

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

2.3 Hoare Logic (StateMonad)

```
named_theorems wprule

definition
  valid :: "('s ⇒ bool) ⇒ ('a,'e,'s) state_monad ⇒
             ('a ⇒ 's ⇒ bool) ⇒
             ('e ⇒ bool) ⇒ bool"
  (‹⦃_⦄/ _ /(⦃_⦄,/ ⦃_⦄)›)
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⟩

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 wpmodify[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 wpoption[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 wpprod[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 wpvcg = 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 :: "(int ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ int ⇒ 'b"
where
  "iter f v x = (if x ≤ 0 then v
                 else f (x-1) (iter f v (x-1)))"

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

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


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


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

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) < 2^b"
⟨proof⟩

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

```
lemma upper_bound3:
  fixes b::nat
    and v::int
      defines "x ≡ - (2 ^ (b - 1)) + (v + 2 ^ (b - 1)) mod 2 ^ b"
    assumes "b>0"
    shows "x < 2^(b-1)"
  ⟨proof⟩

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

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

lemma createSInt_id_g0:
    fixes b::nat
      and v::int
  assumes "v ≥ 0"
      and "v < 2^(b-1)"
      and "b > 0"
    shows "createSInt b v = ShowL_int v"
⟨proof⟩

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

lemma createSInt_id:
    fixes b::nat
      and v::int
  assumes "v < 2^(b-1)"
      and "v ≥ -(2^(b-1))"
      and "b > 0"
    shows "createSInt b v = ShowL_int v" ⟨proof⟩


definition createUInt :: "nat ⇒ int ⇒ Valuetype"
  where "createUInt b v = ShowL_int (v mod (2^b))"

declare createUInt_def[solidity_symbex]

lemma createUInt_id:
  assumes "v ≥ 0"
      and "v < 2^b"
    shows "createUInt b v =  ShowL_int v"
⟨proof⟩

definition createBool :: "bool ⇒ Valuetype"
where
  "createBool b = ShowL_bool b"
```

```
declare createBool_def [solidity_symbex]

definition createAddress :: "Address ⇒ Valuetype"
where
  "createAddress ad = ad"

declare createAddress_def [solidity_symbex]

definition checkSInt :: "nat ⇒ Valuetype ⇒ bool"
where
  "checkSInt b v = ((foldr (∧) (map is_digit (String.explode v)) True) ∧(ReadL_int v ≥ -(2^(b-1)) ∧
ReadL_int v < 2^(b-1)))"

declare checkSInt_def [solidity_symbex]

definition checkUInt :: "nat ⇒ Valuetype ⇒ bool"
where
  "checkUInt b v = ((foldr (∧) (map is_digit (String.explode v)) True) ∧ (ReadL_int v ≥ 0 ∧ ReadL_int v
< 2^b))"
declare checkUInt_def  [solidity_symbex]

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

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 b2 < b1
      then Some (createSInt b1 (op ⌈v1⌉ ⌈v2⌉), TSInt b1)
      else None)"
| "olift op (TUInt b1) (TSInt b2) v1 v2 =
    (if b1 < b2
      then Some (createSInt b2 (op ⌈v1⌉ ⌈v2⌉), TSInt b2)
      else None)"
| "olift _ _ _ _ _ = None"


fun plift ::
```

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


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


definition sub :: "Types ⇒ Types ⇒ Valuetype ⇒ Valuetype ⇒ (Valuetype * Types) option"
where
  "sub = olift (-)"


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


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


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 = ShowL_bool True ∧ b = ShowL_bool True then Some (ShowL_bool True, TBool)
    else Some (ShowL_bool False, TBool))"
| "vtand _ _ _ _ = None"


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

definition checkBool :: "Valuetype  ⇒ bool"
where
  "checkBool v = (if (v = STR ''True'' ∨ v = STR ''False'') then True else False)"

declare checkBool_def [solidity_symbex]
```

```
definition checkAddress :: "Valuetype  ⇒ bool"
  where
    "checkAddress v = (if (size v = 42 ∧ ((String.explode v !1) = CHR ''x'')) then True else False)"

declare checkAddress_def [solidity_symbex]
```

```
primrec ival :: "Types ⇒ Valuetype"
where
  "ival (TSInt x) = ShowL_int 0"
| "ival (TUInt x) = ShowL_int 0"
| "ival TBool = ShowL_bool False"
| "ival TAddr = STR ''0x0000000000000000000000000000000000000000''"
```

```
declare convert.simps [simp del, solidity_symbex add]
declare olift.simps [simp del, solidity_symbex add]
declare plift.simps [simp del, solidity_symbex add]
declare vtand.simps [simp del, solidity_symbex add]
declare vtor.simps [simp del, solidity_symbex add]

end
```

3.2 Accounts (Accounts)

```
theory Accounts
imports Valuetypes
begin

type_synonym Balance = Valuetype
type_synonym Identifier = String.literal
```

```
datatype atype =
    EOA
  | Contract Identifier

record account =
  bal :: Balance
  type :: "atype option"
  contracts :: nat

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

definition emptyAcc :: account
  where "emptyAcc = (|bal = ShowL_int 0, type = None, contracts = 0|)"

declare emptyAcc_def [solidity_symbex]

type_synonym Accounts = "Address ⇒ account"

definition emptyAccount :: "Accounts"
where
```

```
  "emptyAccount _ = emptyAcc"

declare emptyAccount_def [solidity_symbex]

definition addBalance :: "Address ⇒ Valuetype ⇒ Accounts ⇒ Accounts option"
where
  "addBalance ad val acc =
    (if ReadL_int val ≥ 0
      then (let v = ReadL_int (bal (acc ad)) + ReadL_int val
         in if (v < 2^256)
           then Some (acc(ad := acc ad (|bal:=ShowL_int v|)))
           else None)
      else None)"

declare addBalance_def [solidity_symbex]

lemma addBalance_val1:
  assumes "addBalance ad val acc = Some acc'"
    shows "ReadL_int val ≥ 0"
⟨proof⟩

lemma addBalance_val2:
  assumes "addBalance ad val acc = Some acc'"
    shows "ReadL_int (bal (acc ad)) + ReadL_int val < 2^256"
⟨proof⟩

lemma addBalance_limit:
  assumes "addBalance ad val acc = Some acc'"
     and "∀ad. ReadL_int (bal (acc ad)) ≥ 0 ∧ ReadL_int (bal (acc ad)) < 2 ^ 256"
   shows "∀ad. ReadL_int (bal (acc' ad)) ≥ 0 ∧ ReadL_int (bal (acc' ad)) < 2 ^ 256"
⟨proof⟩

lemma addBalance_add:
  assumes "addBalance ad val acc = Some acc'"
    shows "ReadL_int (bal (acc' ad)) = ReadL_int (bal (acc ad)) + ReadL_int val"
⟨proof⟩

lemma addBalance_mono:
  assumes "addBalance ad val acc = Some acc'"
    shows "ReadL_int (bal (acc' ad)) ≥ ReadL_int (bal (acc ad))"
⟨proof⟩

lemma addBalance_eq:
  assumes "addBalance ad val acc = Some acc'"
      and "ad ≠ ad'"
    shows "bal (acc ad') = bal (acc' ad')"
⟨proof⟩

definition subBalance :: "Address ⇒ Valuetype ⇒ Accounts ⇒ Accounts option"
  where
  "subBalance ad val acc =
    (if ReadL_int val ≥ 0
      then (let v = ReadL_int (bal (acc ad)) - ReadL_int val
         in if (v ≥ 0)
           then Some (acc(ad := acc ad(|bal:=ShowL_int v|)))
           else None)
      else None)"

declare subBalance_def [solidity_symbex]

lemma subBalance_val1:
  assumes "subBalance ad val acc = Some acc'"
    shows "ReadL_int val ≥ 0"
⟨proof⟩
```

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

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

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

lemma subBalance_mono:
  assumes "subBalance ad val acc = Some acc'"
    shows "ReadL_int (bal (acc ad)) ≥ ReadL_int (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 "ReadL_int val ≥ 0"
⟨proof⟩

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

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

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

lemma transfer_sub:
  assumes "transfer ads addr val acc = Some acc'"
      and "addr ≠ ads"
  shows "ReadL_int (bal (acc' ads)) = ReadL_int (bal (acc ads)) - ReadL_int val"
⟨proof⟩
```

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

lemma transfer_mono:
  assumes "transfer ads addr val acc = Some acc'"
  shows "ReadL_int (bal (acc' addr)) ≥ ReadL_int (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. ReadL_int (bal (acc ad)) ≥ 0 ∧ ReadL_int (bal (acc ad)) < 2 ^ 256"
   shows "∀ad. ReadL_int (bal (acc' ad)) ≥ 0 ∧ ReadL_int (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 ⇒ String.literal ⇒ Location"
  where "hash loc ix = ix + (STR ''.'' + loc)"

declare hash_def [solidity_symbex]

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 ''.'' ∉ set (String.explode i)"
    and "CHR ''.'' ∉ set (String.explode i')"
  shows "l = l' ∧ i = i'"
⟨proof⟩
```

4.1.2 General Store

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

definition accessStore :: "Location ⇒ 'v Store ⇒ 'v option"
where "accessStore loc st = fmlookup (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 :: "'v Store ⇒ Location * ('v Store)"
where "allocate s = (let ntop = Suc(toploc s) in (ShowL_nat ntop, s (|toploc := ntop|)))"


definition updateStore :: "Location ⇒ 'v ⇒ 'v Store ⇒ 'v Store"
```

```
where "updateStore loc val s = s (| mapping := fmupd loc val (mapping s)|)"

declare updateStore_def [solidity_symbex]

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

declare push_def [solidity_symbex]
```

4.1.3 Stack

```
datatype 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 STypes = STArray int 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 ⇒ Location ⇒ StorageT ⇒ Storagevalue"
where
  "accessStorage t loc sto =
    (case sto $$ loc of
      Some v ⇒ v
    | None ⇒ ival t)"

declare accessStorage_def [solidity_symbex]
```

Copy from storage to storage

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

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

declare copy_def [solidity_symbex]
```

```
abbreviation mystorage2::StorageT
where "mystorage2 ≡ (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

type_synonym MemoryT = "Memoryvalue Store"

type_synonym CalldataT = MemoryT

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

Example

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

Initialization

Definition

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

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

declare minit_def [solidity_symbex]
```

Example

```
lemma "minit 2 (MTArray 2 (MTValue TBool)) emptyStore =
(|mapping = fmap_of_list
  [(STR ''0.0'', MPointer STR ''0.0''), (STR ''0.0.0'', MValue STR ''False''),
```

```
  (STR ''1.0.0'', MValue STR ''False''), (STR ''1.0'', MPointer STR ''1.0''),
  (STR ''0.1.0'', MValue STR ''False''), (STR ''1.1.0'', MValue STR ''False'')],
 toploc = 1⦈" ⟨proof⟩
```

Copy from memory to memory

Definition

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

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

declare cpm2m_def [solidity_symbex]
```

Example

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

abbreviation mymemory2::MemoryT
  where "mymemory2 ≡
    ⦇mapping = fmap_of_list
      [(STR ''0.5'', MValue STR ''True''),
       (STR ''1.5'', MValue STR ''False'')],
     toploc = 0⦈"

lemma "cpm2m (STR ''1.0'') (STR ''5'') 2 (MTValue TBool) mymemory emptyStore = Some mymemory2" ⟨proof⟩
```

4.1.6 Copy from storage to memory

Definition

```
primrec cps2mrec :: "Location ⇒ Location ⇒ STypes ⇒ StorageT ⇒ MemoryT ⇒ MemoryT option"
where
  "cps2mrec locs locm (STArray x t) sto mem =
    (let m = updateStore locm (MPointer locm) mem
    in iter' (λi m'. cps2mrec (hash locs (ShowL_int i)) (hash locm (ShowL_int i)) t sto m') m x)"
| "cps2mrec locs locm (STValue t) sto mem =
    (let v = accessStorage t locs sto
    in Some (updateStore locm (MValue v) mem))"
| "cps2mrec _ _ (STMap _ _) _ _ = None"

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

declare cps2m_def [solidity_symbex]
```

Example

```
abbreviation mystorage3::StorageT
where "mystorage3 ≡ (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
emptyStore)) = Some mymemory"
  ⟨proof⟩
```

4.1.7 Copy from memory to storage

Definition

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

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

declare cpm2s_def [solidity_symbex]
```

Example

```
lemma "cpm2s (STR ''0'') (STR ''1'') 2 (MTArray 2 (MTValue 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 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"
  ⟨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 ⇒ Type ⇒ Denvalue ⇒ Environment ⇒ Environment"
  where "updateEnv i t v e = e (| denvalue := fmupd i (t,v) (denvalue e) |)"

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

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

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

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

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

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

⟨proof⟩

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

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

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

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

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

Examples

```
abbreviation "myenv::Environment ≡ eempty ⦇denvalue := fmupd STR ''id1'' (Value TBool, Stackloc STR
''0'') fmempty⦈"
abbreviation "mystack::Stack ≡ ⦇mapping = fmupd (STR ''0'') (KValue STR ''True'') fmempty, toploc = 1⦈"

abbreviation "myenv2::Environment ≡ eempty ⦇denvalue := fmupd STR ''id2'' (Value TBool, Stackloc STR
''1'') (fmupd STR ''id1'' (Value TBool, Stackloc STR ''0'') fmempty)⦈"
abbreviation "mystack2::Stack ≡ ⦇mapping = fmupd (STR ''1'') (KValue STR ''False'') (fmupd (STR ''0'')
(KValue STR ''True'') fmempty), 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 ⇒
(Address ⇒ StorageT)
    ⇒ CalldataT × MemoryT × Stack × Environment ⇒ (CalldataT × MemoryT × Stack × Environment)
option"
  where

  "decl i (Value t) None _ _ _ _ (c, m, k, e) = Some (c, m, (astack 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)
    (λv'. Some (c, m, astack i (Value t) (KValue v') (k, e)))"
| "decl _ (Value _) _ _ _ _ _ _ = None"


| "decl i (Calldata (MTArray x t)) (Some (KCDptr p, _)) True cd _ _ (c, m, k, e) =
    (let l = ShowL_nat (toploc c);
         (_, c') = allocate c
     in Option.bind (cpm2m p l x t cd c')
      (λc''. Some (c'', m, astack i (Calldata (MTArray x t)) (KCDptr l) (k, e))))"
| "decl i (Calldata (MTArray x t)) (Some (KMemptr p, _)) True _ mem _ (c, m, k, e) =
    (let l = ShowL_nat (toploc c);
         (_, c') = allocate c
     in Option.bind (cpm2m p l x t mem c')
      (λc''. Some (c'', m, astack i (Calldata (MTArray x t)) (KCDptr l) (k, e))))"
| "decl i (Calldata _) _ _ _ _ _ _ = None"


| "decl i (Memory (MTArray x t)) None _ _ _ _ (c, m, k, e) =
    (let m' = minit x t m
      in Some (c, m', astack i (Memory (MTArray x t)) (KMemptr (ShowL_nat (toploc m))) (k, e)))"
| "decl i (Memory (MTArray x t)) (Some (KMemptr p, _)) True _ mem _ (c, m, k, e) =
   Option.bind (cpm2m p (ShowL_nat (toploc m)) x t mem (snd (allocate m)))
    (λm'. Some (c, m', astack i (Memory (MTArray x t)) (KMemptr (ShowL_nat (toploc m))) (k, e)))"
| "decl i (Memory (MTArray x t)) (Some (KMemptr p, _)) False _ _ _ (c, m, k, e) =
   Some (c, m, astack i (Memory (MTArray x t)) (KMemptr p) (k, e))"
| "decl i (Memory (MTArray x t)) (Some (KCDptr p, _)) _ cd _ _ (c, m, k, e) =
   Option.bind (cpm2m p (ShowL_nat (toploc m)) x t cd (snd (allocate m)))
    (λm'. Some (c, m', astack i (Memory (MTArray x t)) (KMemptr (ShowL_nat (toploc m))) (k, e)))"
| "decl i (Memory (MTArray x t)) (Some (KStoptr p, Storage (STArray x' t'))) _ _ _ s (c, m, k, e) =
   Option.bind (cps2m p (ShowL_nat (toploc m)) x' t' (s (address e)) (snd (allocate m)))
    (λm''. Some (c, m'', astack i (Memory (MTArray x t)) (KMemptr (ShowL_nat (toploc m))) (k, e)))"
| "decl _ (Memory _) _ _ _ _ _ _ = None"


| "decl i (Storage (STArray x t)) (Some (KStoptr p, _)) _ _ _ _ (c, m, k, e) =
    Some (c, m, astack i (Storage (STArray x t)) (KStoptr p) (k, e))"
| "decl i (Storage (STMap t t')) (Some (KStoptr p, _)) _ _ _ _ (c, m, k, e) =
   Some (c, m, astack i (Storage (STMap t t')) (KStoptr p) (k, e))"
| "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' ∧ (∀x. x ≠
a1 ⟶ fmlookup (denvalue env') x = fmlookup (denvalue env) x)"
  ⟨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 nat int
           | UINT nat int
           | ADDRESS String.literal
           | BALANCE E
           | THIS
           | SENDER
           | VALUE
           | TRUE
           | FALSE
           | LVAL L
           | PLUS E E
           | MINUS E E
           | EQUAL E E
           | LESS E E
           | AND E E
           | OR E E
           | NOT E
           | CALL Identifier "E list"
           | ECALL E Identifier "E list"
           | 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

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

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

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

5.1.2 State

```
type_synonym Gas = nat

record State =
  accounts :: Accounts
  stack :: Stack
  memory :: MemoryT
  storage :: "Address ⇒ StorageT"
  gas :: Gas

lemma all_gas_le:
  assumes "gas x<(gas y::nat)"
      and "∀z. gas z < gas y ⟶ P z ⟶ Q z"
    shows "∀z. gas z ≤ gas x ∧ P z ⟶ Q z" ⟨proof⟩

lemma all_gas_less:
  assumes "∀z. gas z < gas y ⟶ P z"
      and "gas x≤(gas y::nat)"
    shows "∀z. gas z < 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 = emptyStore,
    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 beeing 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 Environment_P = "(Identifier, Contract) fmap"

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

declare init_def [solidity_symbex]

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

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

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

```
  ⟨proof⟩

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

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

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

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

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

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

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

```
definition ffold_init::"(String.literal, Member) fmap ⇒ Environment ⇒ String.literal fset ⇒
Environment" where
          ‹ffold_init ct a c = ffold (init ct) a c›
declare ffold_init_def [simp,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 "⋀v. x = KValue v ⟹ f v s = f' v s"
      and "⋀p. x = KCDptr p ⟹ g p s = g' p s"
      and "⋀p. x = KMemptr p ⟹ h p s = h' p s"
      and "⋀p. x = KStoptr p ⟹ i p s = i' p s"
    shows "(case x of KValue v ⇒ f v | KCDptr p ⇒ g p | KMemptr p ⇒ h p | KStoptr p ⇒ i p) s
         = (case x' of KValue v ⇒ f' v | KCDptr p ⇒ g' p | KMemptr p ⇒ h' p | KStoptr p ⇒ i' p) s"
  ⟨proof⟩

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

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

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

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

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

lemma bind_case_contract_cong [fundef_cong]:
  assumes "x = x'"
      and "⋀a. x = Method a ⟹ f a s = f' a s"
      and "⋀a. x = 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, solidity_symbex]

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 LType = LStackloc Location
               | LMemloc Location
               | LStoreloc Location

locale expressions_with_gas =
  fixes costs_e :: "E ⇒ Environment ⇒ CalldataT ⇒ State ⇒ Gas"
    and ep::Environment_P
  assumes call_not_zero[termination_simp]: "⋀e cd st i ix. 0 < (costs_e (CALL i ix) e cd st)"
      and ecall_not_zero[termination_simp]: "⋀e cd st a i ix. 0 < (costs_e (ECALL a i ix) e cd st)"
begin
function (domintros) msel::"bool ⇒ MTypes ⇒ Location ⇒ E list ⇒ Environment ⇒ CalldataT ⇒ State
⇒ (Location * MTypes, Ex, Gas) state_monad"
     and ssel::"STypes ⇒ Location ⇒ E list ⇒ Environment ⇒ CalldataT ⇒ State ⇒ (Location *
STypes, Ex, Gas) state_monad"
     and expr::"E ⇒ Environment ⇒ CalldataT ⇒ State ⇒ (Stackvalue * Type, Ex, Gas) state_monad"
     and load :: "bool ⇒ (Identifier × Type) list ⇒ E list ⇒ Environment ⇒ CalldataT ⇒ Stack ⇒
MemoryT ⇒ Environment ⇒ CalldataT ⇒ State ⇒ (Environment × CalldataT × Stack × MemoryT, Ex, Gas)
```

```
state_monad"
     and rexp::"L ⇒ Environment ⇒ CalldataT ⇒ State ⇒ (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 ← expr x env cd st;
      (v, t') ← case kv of (KValue v, Value t') ⇒ return (v, t') | _ ⇒ throw Err;
      assert Err (λ_. less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool));
      return (hash loc v, t)
    }) g"

| "msel mm (MTArray al t) loc (x # y # ys) env cd st g =
    (do {
      kv ← expr x env cd st;
      (v, t') ← case kv of (KValue v, Value t') ⇒ return (v,t') | _ ⇒ throw Err;
      assert Err (λ_. less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool));
      l ← case accessStore (hash loc v) (if mm then memory st else cd) of Some (MPointer l) ⇒ return l
| _ ⇒ 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 ← expr x env cd st;
      (v, t') ← case kv of (KValue v, Value t') ⇒ return (v, t') | _ ⇒ throw Err;
      assert Err (λ_. less t' (TUInt 256) v (ShowL_int al) = Some (ShowL_bool True, TBool));
      ssel t (hash loc v) xs env cd st
    }) g"
| "ssel (STMap _ t) loc (x # xs) env cd st g =
    (do {
      kv ← expr x env cd st;
      v ← case kv of (KValue v, _) ⇒ return v | _ ⇒ throw Err;
      ssel t (hash loc v) xs env cd st
    }) g"
| "expr (E.INT b x) e cd st g =
    (do {
      assert Gas (λg. g > costs_e (E.INT b x) e cd st);
      modify (λg. g - costs_e (E.INT b x) e cd st);
      assert Err (λ_. b ∈ vbits);
      return (KValue (createSInt b x), Value (TSInt b))
    }) g"
| "expr (UINT b x) e cd st g =
    (do {
      assert Gas (λg. g > costs_e (UINT b x) e cd st);
      modify (λg. g - costs_e (UINT b x) e cd st);
      assert Err (λ_. b ∈ vbits);
      return (KValue (createUInt b x), Value (TUInt b))
  }) g"
| "expr (ADDRESS ad) e cd st g =
    (do {
      assert Gas (λg. g > costs_e (ADDRESS ad) e cd st);
      modify (λg. g - costs_e  (ADDRESS ad) e cd st);
      return (KValue ad, Value TAddr)
    }) g"
| "expr (BALANCE ad) e cd st g =
    (do {
      assert Gas (λg. g > costs_e (BALANCE ad) e cd st);
      modify (λg. g - costs_e  (BALANCE ad) e cd st);
      kv ← expr ad e cd st;
      adv ← case kv of (KValue adv, Value TAddr) ⇒ return adv | _ ⇒ throw Err;
      return (KValue (bal ((accounts st) adv)), Value (TUInt 256))
```

```
  }) g"
| "expr THIS e cd st g =
  (do {
    assert Gas (λg. g > costs_e THIS e cd st);
    modify (λg. g - costs_e THIS e cd st);
    return (KValue (address e), Value TAddr)
  }) g"
| "expr SENDER e cd st g =
  (do {
    assert Gas (λg. g > costs_e SENDER e cd st);
    modify (λg. g - costs_e SENDER e cd st);
    return (KValue (sender e), Value TAddr)
  }) g"
| "expr VALUE e cd st g =
  (do {
    assert Gas (λg. g > costs_e VALUE e cd st);
    modify (λg. g - costs_e VALUE e cd st);
    return (KValue (svalue e), Value (TUInt 256))
  }) g"
| "expr TRUE e cd st g =
  (do {
    assert Gas (λg. g > costs_e TRUE e cd st);
    modify (λg. g - costs_e TRUE e cd st);
    return (KValue (ShowL_bool True), Value TBool)
  }) g"
| "expr FALSE e cd st g =
  (do {
    assert Gas (λg. g > costs_e FALSE e cd st);
    modify (λg. g - costs_e FALSE e cd st);
    return (KValue (ShowL_bool False), Value TBool)
  }) g"
| "expr (NOT x) e cd st g =
  (do {
    assert Gas (λg. g > costs_e (NOT x) e cd st);
    modify (λg. g - costs_e (NOT x) e cd st);
    kv ← expr x e cd st;
    v ← case kv of (KValue v, Value TBool) ⇒ return v | _ ⇒ throw Err;
    (if v = ShowL_bool True then expr FALSE e cd st
     else if v = ShowL_bool False then expr TRUE e cd st
     else throw Err)
  }) g"
| "expr (PLUS e1 e2) e cd st g =
  (do {
    assert Gas (λg. g > costs_e (PLUS e1 e2) e cd st);
    modify (λg. 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 (λg. g > costs_e (MINUS e1 e2) e cd st);
    modify (λg. 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 (λg. g > costs_e (LESS e1 e2) e cd st);
    modify (λg. 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 (λg. g > costs_e (EQUAL e1 e2) e cd st);
    modify (λg. 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 (λg. g > costs_e (AND e1 e2) e cd st);
      modify (λg. 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 (λg. g > costs_e (OR e1 e2) e cd st);
      modify (λg. 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 (λg. g > costs_e (LVAL i) e cd st);
      modify (λg. 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 (λg. g > costs_e (CALL i xe) e cd st);
      modify (λg. g - costs_e (CALL i xe) e cd st);
      (ct, _) ← option Err (λ_. ep $$ (contract e));
      (fp, x) ← case ct $$ i of Some (Function (fp, False, x)) ⇒ return (fp, x) | _ ⇒ throw Err;
      let e' = ffold_init ct (emptyEnv (address e) (contract e) (sender e) (svalue e)) (fmdom ct);
      (e_l, cd_l, k_l, m_l) ← load False fp xe e' emptyStore emptyStore (memory st) e cd st;
      expr x e_l cd_l (st⦇stack:=k_l, memory:=m_l⦈)
    }) g"

| "expr (ECALL ad i xe) e cd st g =
    (do {
      assert Gas (λg. g > costs_e (ECALL ad i xe) e cd st);
      modify (λg. g - costs_e (ECALL ad i xe) e cd st);
      kad ← expr ad e cd st;
      adv ← case kad of (KValue adv, Value TAddr) ⇒ return adv | _ ⇒ throw Err;
      assert Err (λ_. adv ≠ address e);
      c ← case type (accounts st adv) of Some (Contract c) ⇒ return c | _ ⇒ throw Err;
      (ct, _) ← option Err (λ_. ep $$ c);
      (fp, x) ← case ct $$ i of Some (Function (fp, True, x)) ⇒ return (fp, x) | _ ⇒ throw Err;
      let e' = ffold_init ct (emptyEnv adv c (address e) (ShowL_nat 0)) (fmdom ct);
      (e_l, cd_l, k_l, m_l) ← load True fp xe e' emptyStore emptyStore emptyStore e cd st;
      expr x e_l cd_l (st⦇stack:=k_l, memory:=m_l⦈)
    }) g"
| "load cp ((i_p, t_p)#pl) (ex#el) e_v' cd' sck' mem' e_v cd st g =
    (do {
      (v, t) ← expr ex e_v cd st;
      (c, m, k, e) ← case decl i_p t_p (Some (v,t)) cp cd (memory st) (storage st) (cd', mem',  sck',
e_v') of Some (c, m, k, e) ⇒ return (c, m, k, e) | None ⇒ throw Err;
      load cp pl el e c k m e_v cd st
    }) g"
| "load _ [] (_#_) _ _ _ _ _ _ _ g = throw Err g"
| "load _ (_#_) [] _ _ _ _ _ _ _ g = throw Err g"
| "load _ [] [] e_v' cd' sck' mem' e_v cd st g = return (e_v', 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 (Storage (STValue t), Storeloc l) ⇒ return (KValue (accessStorage t l (storage st (address
e))), Value t)
    | Some (Storage (STArray x t), Storeloc l) ⇒ return (KStoptr l, 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 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, Memory (MTArray x t''))
                  }
              )
            }
        | Some (KStoptr l') ⇒
            do {
              t ← case tp of 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'', Storage t')
              | STMap _ _   ⇒ return (KStoptr l'', Storage t'))
             }
        | _ ⇒ throw Err)
    | Some (tp, (Storeloc l)) ⇒
          do {
            t ← case tp of 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', Storage t')
            | STMap _ _   ⇒ return (KStoptr l', Storage t'))
```

```
        }
    | None ⇒ throw Err) g"
| "expr CONTRACTS e cd st g =
    (do {
      assert Gas (λg. g > costs_e CONTRACTS e cd st);
      modify (λg. g - costs_e CONTRACTS e cd st);
      prev ← case contracts (accounts st (address e)) of 0 ⇒ throw Err | Suc n ⇒ return n;
      return (KValue (hash (address e) (ShowL_nat 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)"
    "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 (snd l)))"
      | "mgas (Inl (Inr l)) = snd (snd (snd (snd (snd (snd l))))))"
      | "mgas (Inl (Inl l)) = snd (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]

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

lemma *lexp_gas[rule_format]:*
```
    "∀ l5' t5' g5'. lexp l5 ev5 cd5 st5 g5 = Normal ((l5', t5'), g5') ⟶ g5' ≤ 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 (gas s) of
                    Normal (a,g) ⇒ Normal(a,s⦇gas:=g⦈)
                  | Exception e ⇒ Exception e)"
```

lemma *wptoState[wprule]:*
```
  assumes "⋀a g. gm s (gas s) = Normal (a, g) ⟹ P a (s⦇gas:=g⦈)"
      and "⋀e. gm s (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. gas st > costs SKIP e cd st);
      modify (λst. st⦇gas := gas st - costs SKIP e cd st⦈)
    }) st"
| "stmt (ASSIGN lv ex) env cd st =
    (do {
      assert Gas (λst. gas st > costs (ASSIGN lv ex) env cd st);
      modify (λst. st⦇gas := gas st - costs (ASSIGN lv ex) env cd st⦈);
      re ← toState (expr ex env cd);
      case re of
        (KValue v, Value t) ⇒
```

```
          do {
            rl ← toState (lexp lv env cd);
            case rl 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, 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, 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 (MTArray x t)) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Memory _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
                  m ← option Err (λst. cpm2m p p' x t cd (memory st));
                  modify (λst. st(|memory := m|))
                }
            | (LStackloc l, Storage _) ⇒
                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)|))
                }
            | (LStoreloc l, _) ⇒
                do {
                  s ← option Err (λst. cpm2s p l x t cd (storage st (address env)));
                  modify (λst. st (|storage := (storage st) (address env := s)|))
                }
            | (LMemloc l, _) ⇒
                do {
                  m ← option Err (λst. cpm2m p l x t cd (memory st));
                  modify (λst. st (|memory := m|))
                }
            | _ ⇒ throw Err
          }
      | (KMemptr p, Memory (MTArray x t)) ⇒
          do {
            rl ← toState (lexp lv env cd);
            case rl of
              (LStackloc l, Memory _) ⇒ modify (λst. st(|stack := updateStore l (KMemptr p) (stack
st)|))
            | (LStackloc l, Storage _) ⇒
                do {
                  sv ← applyf (λst. accessStore l (stack st));
                  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)|))
                }
```

```
          | (LStoreloc l, _) ⇒
              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)⦈)
              }
          | (LMemloc l, _) ⇒ modify (λst. st ⦇memory := updateStore l (MPointer p) (memory st)⦈)
          | _ ⇒ throw Err
        }
    | (KStoptr p, Storage (STArray x t)) ⇒
        do {
          rl ← toState (lexp lv env cd);
          case rl of
            (LStackloc l, Memory _) ⇒
              do {
                sv ← applyf (λst. accessStore l (stack st));
                p' ← case sv of Some (KMemptr p') ⇒ return p' | _ ⇒ throw Err;
                m ← option Err (λst. cps2m p p' x t (storage st (address env)) (memory st));
                modify (λst. st⦇memory := m⦈)
              }
          | (LStackloc l, Storage _) ⇒ modify (λst. st⦇stack := updateStore l (KStoptr p) (stack
st)⦈)
          | (LStoreloc l, _) ⇒
              do {
                s ← option Err (λst. copy p l x t (storage st (address env)));
                modify (λst. st ⦇storage := (storage st) (address env := s)⦈)
              }
          | (LMemloc l, _) ⇒
              do {
                m ← option Err (λst. cps2m p l x t (storage st (address env)) (memory st));
                modify (λst. st⦇memory := m⦈)
              }
          | _ ⇒ throw Err
        }
    | (KStoptr p, Storage (STMap t t')) ⇒
        do {
          rl ← toState (lexp lv env cd);
          l ← case rl of (LStackloc l, _) ⇒ return l | _ ⇒ throw Err;
          modify (λst. st⦇stack := updateStore l (KStoptr p) (stack st)⦈)
        }
    | _ ⇒ throw Err
  }) st"
| "stmt (COMP s1 s2) e cd st =
  (do {
    assert Gas (λst. gas st > costs (COMP s1 s2) e cd st);
    modify (λst. st⦇gas := 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. gas st > costs (ITE ex s1 s2) e cd st);
    modify (λst. st⦇gas := 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 = ShowL_bool True then stmt s1 e cd
    else if b = ShowL_bool False then stmt s2 e cd
    else throw Err
  }) st"
| "stmt (WHILE ex s0) e cd st =
  (do {
    assert Gas (λst. gas st > costs (WHILE ex s0) e cd st);
    modify (λst. st⦇gas := 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 = ShowL_bool True then
        do {
          stmt s0 e cd;
          stmt (WHILE ex s0) e cd
        }
      else if b = ShowL_bool False then return ()
      else throw Err
    }) st"
| "stmt (INVOKE i xe) e cd st =
    (do {
      assert Gas (λst. gas st > costs (INVOKE i xe) e cd st);
      modify (λst. st⦇gas := 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);
      m_o ← applyf memory;
      (e_l, cd_l, k_l, m_l) ← toState (load False fp xe e' emptyStore emptyStore m_o e cd);
      k_o ← applyf stack;
      modify (λst. st⦇stack:=k_l, memory:=m_l⦈);
      stmt f e_l cd_l;
      modify (λst. st⦇stack:=k_o⦈)
    }) st"



| "stmt (EXTERNAL ad i xe val) e cd st =
    (do {
      assert Gas (λst. gas st > costs (EXTERNAL ad i xe val) e cd st);
      modify (λst. st⦇gas := 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 (Contract c) ⇒ return c st | _ ⇒ throw Err st);
      (ct, _, fb) ← option Err (λ_. ep $$ c);
      kv ← toState (expr val e cd);
      (v, t) ← case kv of (KValue v, Value t) ⇒ return (v, t) | _ ⇒ throw Err;
      v' ← option Err (λ_. convert t (TUInt 256) v);
      let e' = ffold_init ct (emptyEnv adv c (address e) v') (fmdom ct);
      case ct $$ i of
        Some (Method (fp, True, f)) ⇒
          do {
            (e_l, cd_l, k_l, m_l) ← toState (load True fp xe e' emptyStore emptyStore emptyStore e cd);
            acc ← option Err (λst. transfer (address e) adv v' (accounts st));
            (k_o, m_o) ← applyf (λst. (stack st, memory st));
            modify (λst. st⦇accounts := acc, stack:=k_l,memory:=m_l⦈);
            stmt f e_l cd_l;
            modify (λst. st⦇stack:=k_o, memory := m_o⦈)
          }
      | None ⇒
          do {
            acc ← option Err (λst. transfer (address e) adv v' (accounts st));
            (k_o, m_o) ← applyf (λst. (stack st, memory st));
            modify (λst. st⦇accounts := acc,stack:=emptyStore, memory:=emptyStore⦈);
            stmt fb e' emptyStore;
            modify (λst. st⦇stack:=k_o, memory := m_o⦈)
          }
      | _ ⇒ throw Err
    }) st"
| "stmt (TRANSFER ad ex) e cd st =
    (do {
      assert Gas (λst. gas st > costs (TRANSFER ad ex) e cd st);
      modify (λst. st⦇gas := 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 256) v);
      acc ← applyf accounts;
      case type (acc adv) of
        Some (Contract c) ⇒
          do {
            (ct, _, f) ← option Err (λ_. ep $$ c);
            let e' = ffold_init ct (emptyEnv adv c (address e) v') (fmdom ct);
            (k_o, m_o) ← 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:=emptyStore|));
            stmt f e' emptyStore;
            modify (λst. st(|stack:=k_o, memory := m_o|))
          }
      | 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) e_v cd st =
    (do {
      assert Gas (λst. gas st > costs (BLOCK ((id0, tp), None) s) e_v cd st);
      modify (λst. st(|gas := gas st - costs (BLOCK ((id0, tp), None) s) e_v cd st|));
      (cd', mem', sck', e') ← option Err (λst. decl id0 tp None False cd (memory st) (storage st) (cd,
memory st, stack st, e_v));
      modify (λst. st(|stack := sck', memory := mem'|));
      stmt s e' cd'
    }) st"
| "stmt (BLOCK ((id0, tp), Some ex') s) e_v cd st =
    (do {
      assert Gas (λst. gas st > costs(BLOCK ((id0, tp), Some ex') s) e_v cd st);
      modify (λst. st(|gas := gas st - costs (BLOCK ((id0, tp), Some ex') s) e_v cd st|));
      (v, t) ← toState (expr ex' e_v cd);
      (cd', mem', sck', e') ← option Err (λst. decl id0 tp (Some (v, t)) False cd (memory st) (storage
st) (cd, memory st, stack st, e_v));
      modify (λst. st(|stack := sck', memory := mem'|));
      stmt s e' cd'
    }) st"

| "stmt (NEW i xe val) e cd st =
    (do {
      assert Gas (λst. gas st > costs (NEW i xe val) e cd st);
      modify (λst. st(|gas := gas st - costs (NEW i xe val) e cd st|));
      adv ← applyf (λst. hash (address e) (ShowL_nat (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;
      (ct, cn, _) ← option Err (λ_. ep $$ i);
      let e' = ffold_init ct (emptyEnv adv i (address e) v) (fmdom ct);
      (e_l, cd_l, k_l, m_l) ← toState (load True (fst cn) xe e' emptyStore emptyStore emptyStore e cd);
      modify (λst. st(|accounts := (accounts st)(adv := (|bal = ShowL_int 0, type = Some (Contract i),
contracts = 0|)), storage:=(storage st)(adv := {$$})|));
      acc ← option Err (λst. transfer (address e) adv v (accounts st));
      (k_o, m_o) ← applyf (λst. (stack st, memory st));
      modify (λst. st(|accounts := acc, stack:=k_l, memory:=m_l|));
      stmt (snd cn) e_l cd_l;
      modify (λst. st(|stack:=k_o, memory := m_o|));
      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') ⟶ gas st6' ≤
gas st6)"
⟨proof⟩
```

5.3.4 Termination function

Now we can prove termination using the lemma above.

```
fun sgas
  where "sgas l = 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 "gas st > costs SKIP ev cd st"
      and "st' = st⦇gas := 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⦇gas := gas st - costs  (ASSIGN lv ex) ev cd st⦈) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KValue v, Value t), g)"
      and "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, Value t'),g')"
      and "convert t t' v = Some v'"
      and "st' = st⦇gas := g', stack := updateStore l (KValue v') (stack st)⦈"
```

```
  | (2) v t g l t' g' v'
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KValue v, Value t), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LStoreloc l, Storage (STValue t')),g')"
      and "convert t t' v = Some v'"
      and "st' = st(|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(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KValue v, Value t), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LMemloc l, Memory (MTValue t')),g')"
      and "convert t t' v = Some v'"
      and "st' = st(|gas := g', memory := updateStore l (MValue v') (memory st)|)"
  | (4) p x t g l t' g' p' m
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LStackloc l, Memory t'),g')"
      and "accessStore l (stack st) = Some (KMemptr p')"
      and "cpm2m p p' x t cd (memory st) = Some m"
      and "st' = st(|gas := g', memory := m|)"
  | (5) p x t g l t' g' p' s
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LStackloc l, Storage t'),g')"
      and "accessStore l (stack st) = Some (KStoptr p')"
      and "cpm2s p p' x t cd (storage st (address ev)) = Some s"
      and "st' = st(|gas := g', storage := (storage st) (address ev := s)|)"
  | (6) p x t g l t' g' s
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LStoreloc l, t'),g')"
      and "cpm2s p l x t cd (storage st (address ev)) = Some s"
      and "st' = st(|gas := g', storage := (storage st) (address ev := s)|)"
  | (7) p x t g l t' g' m
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KCDptr p, Calldata (MTArray x t)), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LMemloc l, t'),g')"
      and "cpm2m p l x t cd (memory st) = Some m"
      and "st' = st(|gas := g', memory := m|)"
  | (8) p x t g l t' g'
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KMemptr p, Memory (MTArray x t)), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LStackloc l, Memory t'),g')"
      and "st' = st(|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(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KMemptr p, Memory (MTArray x t)), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LStackloc l, Storage t'),g')"
      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(|gas := g', storage := (storage st) (address ev := s)|)"
  | (10) p x t g l t' g' s
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KMemptr p, Memory (MTArray x t)), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LStoreloc l, t'),g')"
      and "cpm2s p l x t (memory st) (storage st (address ev)) = Some s"
      and "st' = st(|gas := g', storage := (storage st) (address ev := s)|)"
  | (11) p x t g l t' g'
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KMemptr p, Memory (MTArray x t)), g)"
      and "lexp lv ev cd (st(|gas := g|)) g = Normal((LMemloc l, t'),g')"
      and "st' = st(|gas := g', memory := updateStore l (MPointer p) (memory st)|)"
  | (12) p x t g l t' g' p' m
    where "expr ex ev cd (st(|gas := gas st - costs  (ASSIGN lv ex) ev cd st|)) (gas st - costs  (ASSIGN
```

```
lv ex) ev cd st) = Normal ((KStoptr p, Storage (STArray x t)), g)"
      and "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, Memory t'),g')"
      and "accessStore l (stack st) = Some (KMemptr p')"
      and "cps2m p p' x t (storage st (address ev)) (memory st) = Some m"
      and "st' = st⦇gas := g', memory := m⦈"
  | (13) p x t g l t' g'
    where "expr ex ev cd (st⦇gas := gas st - costs  (ASSIGN lv ex) ev cd st⦈) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KStoptr p, Storage (STArray x t)), g)"
      and "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, Storage t'),g')"
      and "st' = st⦇gas := g', stack := updateStore l (KStoptr p) (stack st)⦈"
  | (14) p x t g l t' g' s
    where "expr ex ev cd (st⦇gas := gas st - costs  (ASSIGN lv ex) ev cd st⦈) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KStoptr p, Storage (STArray x t)), g)"
      and "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStoreloc l, t'),g')"
      and "copy p l x t (storage st (address ev)) = Some s"
      and "st' = st⦇gas := g', storage := (storage st) (address ev := s)⦈"
  | (15) p x t g l t' g' m
    where "expr ex ev cd (st⦇gas := gas st - costs  (ASSIGN lv ex) ev cd st⦈) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KStoptr p, Storage (STArray x t)), g)"
      and "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LMemloc l, t'),g')"
      and "cps2m p l x t (storage st (address ev)) (memory st) = Some m"
      and "st' = st⦇gas := g', memory := m⦈"
  | (16) p t t' g l t'' g'
    where "expr ex ev cd (st⦇gas := gas st - costs  (ASSIGN lv ex) ev cd st⦈) (gas st - costs  (ASSIGN
lv ex) ev cd st) = Normal ((KStoptr p, Storage (STMap t t')), g)"
      and "lexp lv ev cd (st⦇gas := g⦈) g = Normal((LStackloc l, t''),g')"
      and "st' = st⦇gas := g', stack := updateStore l (KStoptr p) (stack st)⦈"
⟨proof⟩

lemma comp:
  assumes "stmt (COMP s1 s2) ev cd st = Normal (x, st')"
  obtains (1) st''
  where "gas st > costs (COMP s1 s2) ev cd st"
    and "stmt s1 ev cd (st⦇gas := 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 "gas st > costs (ITE ex s1 s2) ev cd st"
    and "expr ex ev cd (st⦇gas := gas st - costs (ITE ex s1 s2) ev cd st⦈) (gas st - costs (ITE ex s1
s2) ev cd st) = Normal((KValue (ShowL_bool True), Value TBool), g)"
    and "stmt s1 ev cd (st⦇gas := g⦈) = Normal((), st')"
| (False) g
  where "gas st > costs (ITE ex s1 s2) ev cd st"
    and "expr ex ev cd (st⦇gas := gas st - costs (ITE ex s1 s2) ev cd st⦈) (gas st - costs (ITE ex s1
s2) ev cd st) = Normal((KValue (ShowL_bool False), Value TBool), g)"
    and "stmt s2 ev cd (st⦇gas := g⦈) = Normal((), st')"
  ⟨proof⟩

lemma while:
  assumes "stmt (WHILE ex s0) ev cd st = Normal (x, st')"
  obtains (True) g st''
    where "gas st > costs (WHILE ex s0) ev cd st"
      and "expr ex ev cd (st⦇gas := gas st - costs (WHILE ex s0) ev cd st⦈) (gas st - costs (WHILE ex
s0) ev cd st) = Normal((KValue (ShowL_bool True), Value TBool), g)"
      and "stmt s0 ev cd (st⦇gas := g⦈) = Normal((), st'')"
      and "stmt (WHILE ex s0) ev cd st'' = Normal ((), st')"
    | (False) g
  where "gas st > costs (WHILE ex s0) ev cd st"
    and "expr ex ev cd (st⦇gas := gas st - costs (WHILE ex s0) ev cd st⦈) (gas st - costs (WHILE ex s0)
ev cd st) = Normal((KValue (ShowL_bool False), Value TBool), g)"
    and "st' = st⦇gas := g⦈"
```

```
⟨proof⟩

lemma invoke:
  fixes ev
  defines "e' members ≡ ffold (init members) (emptyEnv (address ev) (contract ev) (sender ev) (svalue
ev)) (fmdom members)"
  assumes "stmt (INVOKE i xe) ev cd st = Normal (x, st')"
  obtains ct fb fp f e_l cd_l k_l m_l g st''
    where "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) emptyStore emptyStore (memory (st(|gas := gas st - costs (INVOKE i
xe) ev cd st|))) ev cd (st(|gas := gas st - costs (INVOKE i xe) ev cd st|)) (gas st - costs (INVOKE i xe)
ev cd st) = Normal ((e_l, cd_l, k_l, m_l), g)"
      and "stmt f e_l cd_l (st(|gas:= g, stack:=k_l, memory:=m_l|)) = 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 e_l cd_l k_l m_l g'' acc st''
    where "gas st > costs (EXTERNAL ad' i xe val) ev cd st"
      and "expr ad' ev cd (st(|gas := gas st - costs (EXTERNAL ad' i xe val) ev cd st|)) (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(|gas := g|)) adv) = Some (Contract c)"
      and "ep $$ c = Some (ct, cn, fb')"
      and "expr val ev cd (st(|gas := g|)) g = Normal ((KValue v, Value t), g')"
      and "convert t (TUInt 256) v = Some v'"
      and "fmlookup ct i = Some (Method (fp, True, f))"
      and "load True fp xe (e' ct adv c v') emptyStore emptyStore emptyStore ev cd (st(|gas := g'|)) g' =
Normal ((e_l, cd_l, k_l, m_l), g'')"
      and "transfer (address ev) adv v' (accounts (st(|gas := g''|))) = Some acc"
      and "stmt f e_l cd_l (st(|gas := g'', accounts := acc, stack:=k_l, memory:=m_l|)) = Normal ((), st'')"
      and "st' = st''(|stack:=stack st, memory := memory st|)"
    | (None) adv c g ct cn fb' v t g' v' acc st''
    where "gas st > costs (EXTERNAL ad' i xe val) ev cd st"
      and "expr ad' ev cd (st(|gas := gas st - costs  (EXTERNAL ad' i xe val) ev cd st|)) (gas st - costs
(EXTERNAL ad' i xe val) ev cd st) = Normal ((KValue adv, Value TAddr), g)"
      and "adv ≠ address ev"
      and "type (accounts (st(|gas := g|)) adv) = Some (Contract c)"
      and "ep $$ c = Some (ct, cn, fb')"
      and "expr val ev cd (st(|gas := g|)) g = Normal ((KValue v, Value t), g')"
      and "convert t (TUInt 256) 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') emptyStore (st(|gas := g', accounts := acc, stack:=emptyStore,
memory:=emptyStore|)) = 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 "gas st > costs (TRANSFER ad ex) ev cd st"
      and "expr ad ev cd (st(|gas := gas st - costs (TRANSFER ad ex) ev cd st|)) (gas st - costs
(TRANSFER ad ex) ev cd st) = Normal ((KValue adv, Value TAddr), g)"
      and "expr ex ev cd (st(|gas := g|)) g = Normal ((KValue v, Value t), g')"
      and "convert t (TUInt 256) v = Some v'"
```

```
      and "type (accounts (st⦇gas := g⦈) adv) = Some (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⦇gas := g'⦈) v') emptyStore (st⦇gas := g', accounts := acc,
stack:=emptyStore, memory:=emptyStore⦈) = Normal ((), st'')"
      and "st' = st''⦇stack:=stack st, memory := memory st⦈"
    | (EOA) v t g adv g' v' acc
    where "gas st > costs (TRANSFER ad ex) ev cd st"
      and "expr ad ev cd (st⦇gas := gas st - costs  (TRANSFER ad ex) ev cd st⦈) (gas st - costs
(TRANSFER ad ex) ev cd st) = Normal ((KValue adv, Value TAddr), g)"
      and "expr ex ev cd (st⦇gas := g⦈) g = Normal ((KValue v, Value t), g')"
      and "convert t (TUInt 256) v = Some v'"
      and "type (accounts (st⦇gas := g⦈) adv) = Some EOA"
      and "transfer (address ev) adv v' (accounts st) = Some acc"
      and "st' = st⦇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 "gas st > costs (BLOCK ((id0, tp), None) s) ev cd st"
      and "decl id0 tp None False cd (memory (st⦇gas := gas st - costs (BLOCK ((id0, tp), None) s) ev
cd st⦈)) (storage (st⦇gas := gas st - costs (BLOCK ((id0, tp), None) s) ev cd st⦈)) (cd, memory (st⦇gas
:= gas st - costs (BLOCK ((id0, tp), None) s) ev cd st⦈), stack (st⦇gas := gas st - costs (BLOCK ((id0,
tp), None) s) ev cd st⦈), ev) = Some (cd', mem', sck', e')"
      and "stmt s e' cd' (st⦇gas := 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 "gas st > costs (BLOCK ((id0, tp), Some ex') s) ev cd st"
      and "expr ex' ev cd (st⦇gas := gas st - costs (BLOCK ((id0, tp), Some ex') s) ev cd st⦈) (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⦇gas := g⦈)) (storage (st⦇gas := g⦈)) (cd,
memory (st⦇gas := g⦈), stack (st⦇gas := g⦈), ev) = Some (cd', mem', sck', e')"
      and "stmt s e' cd' (st⦇gas := g, stack := sck', memory := mem'⦈) = Normal ((), st')"
  ⟨proof⟩

lemma new:
  fixes i xe val ev cd st
  defines "st0 ≡ st⦇gas := gas st - costs (NEW i xe val) ev cd st⦈"
  defines "adv0 ≡ hash (address ev) (ShowL_nat (contracts (accounts st0 (address ev))))"
  defines "st1 g ≡ st⦇gas := g, accounts := (accounts st)(adv0 := ⦇bal = ShowL_int 0, type = Some
(Contract i), contracts = 0⦈), storage:=(storage st)(adv0 := {$$})⦈"
  defines "e' members c v ≡ ffold (init members) (emptyEnv adv0 c (address ev) v) (fmdom members)"
  assumes "stmt (NEW i xe val) ev cd st = Normal (x, st')"
  obtains v t g ct cn fb e_l cd_l k_l m_l g' acc st''
    where "gas st > costs (NEW i xe val) ev cd st"
      and "type (accounts st adv0) = None"
      and "expr val ev cd st0 (gas st0) = Normal((KValue v, Value t),g)"
      and "ep $$ i = Some (ct, cn, fb)"
      and "load True (fst cn) xe (e' ct i v) emptyStore emptyStore emptyStore ev cd (st0⦇gas := g⦈) g =
Normal ((e_l, cd_l, k_l, m_l), g')"
      and "transfer (address ev) adv0 v (accounts (st1 g')) = Some acc"
      and "stmt (snd cn) e_l cd_l (st1 g'⦇accounts := acc, stack:=k_l, memory:=m_l⦈) = 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 (UINT _ _) e cd st = 0"
| "costs_ex (ADDRESS _) e cd st = 0"
| "costs_ex (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 fmempty 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 mymemory2::MemoryT
  where "mymemory2 ≡
```

```
    (|mapping = fmap_of_list
      [(STR ''3.2'', MPointer STR ''5'')],
     toploc = 1|)"
```

lemma *"msel True (MTArray 5 (MTArray 6 (MTValue TBool))) (STR ''2'') [UINT 8 3] eempty emptyStore (mystate(|gas:=1|)) 1*
= Normal ((STR ''3.2'', MTArray 6 (MTValue TBool)), 1)" ⟨*proof*⟩

lemma *"msel True (MTArray 5 (MTArray 6 (MTValue TBool))) (STR ''2'') [UINT 8 3, UINT 8 4] eempty emptyStore (mystate(|gas:=1,memory:=mymemory2|)) 1*
= Normal ((STR ''4.5'', MTValue TBool), 1)" ⟨*proof*⟩

lemma *"msel True (MTArray 5 (MTArray 6 (MTValue TBool))) (STR ''2'') [UINT 8 5] eempty emptyStore (mystate(|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 Solidty Evaluator and Code Generator Setup (Solidity_Evaluator)

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

Generalized Unit Tests

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

lemma "STR ''-92134039538802366542421159375273829975''
      = createSInt 128 45648483135649456465465452123894894554654654654654646999465"
  ⟨proof⟩
```

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

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

lemma "STR ''220443428915524155977936330922349307608''
      = (createUInt 128 4564848313564945646546545212389489455465465465465464699946544654654654654168)"
  ⟨proof⟩

lemma "less (TUInt 144) (TSInt 160) (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 dump_Valuetypes::"Types ⇒ Valuetype ⇒ String.literal" where
   "dump_Valuetypes (TSInt _) n = n"
 | "dump_Valuetypes (TUInt _) n = n"
 | "dump_Valuetypes TBool b = (if b = (STR ''True'') then  STR ''true'' else STR ''false'')"
 | "dump_Valuetypes TAddr ad = ad"
```

Memory

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

fun loadRec_Memory :: "Location ⇒ Data_Memory ⇒ MemoryT ⇒ MemoryT" and
        iterateM :: "Location ⇒ MemoryT × nat ⇒ Data_Memory ⇒ MemoryT × nat" where
  "loadRec_Memory loc (MArray dat) mem = fst (foldl (iterateM loc) (updateStore loc (MPointer loc)
mem,0) dat)"
| "loadRec_Memory loc (MBool b) mem = updateStore loc ((MValue ∘ ShowL_bool) b) mem "
| "loadRec_Memory loc (MInt i) mem = updateStore loc ((MValue ∘ ShowL_int) i) mem "
| "loadRec_Memory loc (MAddress ad) mem = updateStore loc (MValue ad) mem"
| "iterateM loc (mem,x) d = (loadRec_Memory (hash loc (ShowL_nat x)) d mem, Suc x)"

definition load_Memory :: "Data_Memory list ⇒ MemoryT ⇒ MemoryT" where
"load_Memory dat mem = (let loc   = ShowL_nat (toploc mem);
                           (m, _) = foldl (iterateM loc) (mem, 0) dat
                        in (snd ∘ allocate) m)"

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

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

Storage

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

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

fun dumpSingle_Storage :: "StorageT ⇒ String.literal ⇒ STypes ⇒ (Location × Location) ⇒
String.literal ⇒ String.literal" where
```

```
"dumpSingle_Storage sto id' tp (loc,l) str =
    (case foldr go_Storage (tolist loc) (str + id', tp) of
      (s, STValue t) ⇒
        (case sto $$ (loc + l) of
           Some v ⇒ s + (STR ''=='') + dump_Valuetypes t v
         | None ⇒ FAILURE)
     | _ ⇒ FAILURE)"
```

definition *iterate* **where**

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

fun *dump_Storage* :: "StorageT ⇒ Location ⇒ String.literal ⇒ STypes ⇒ String.literal ⇒ String.literal" **where**

```
  "dump_Storage sto loc id' (STArray _ t) str = foldl (iterate loc t id' sto) str (locations_map loc
sto)"
| "dump_Storage sto loc id' (STMap _ t) str = foldl (iterate loc t id' sto) str (locations_map loc sto)"
| "dump_Storage sto loc id' (STValue t) str =
    (case sto $$ loc of
      Some v ⇒ str + id' + (STR ''=='') + dump_Valuetypes t v + (STR ''↩'')
    | _ ⇒ str)"
```

fun *loadRec_Storage* :: "Location ⇒ Data_Storage ⇒ StorageT ⇒ StorageT" **and**
iterateSA :: "Location ⇒ StorageT × nat ⇒ Data_Storage ⇒ StorageT × nat" **and**
iterateSM :: "Location ⇒ String.literal × Data_Storage ⇒ StorageT ⇒ StorageT" **where**

```
  "loadRec_Storage loc (SArray dat) sto = fst (foldl (iterateSA loc) (sto,0) dat)"
| "loadRec_Storage loc (SMap dat) sto  = foldr (iterateSM loc) dat sto"
| "loadRec_Storage loc (SBool b) sto = fmupd loc (ShowL_bool b) sto"
| "loadRec_Storage loc (SInt i)  sto = fmupd loc (ShowL_int i) sto"
| "loadRec_Storage loc (SAddress ad) sto = fmupd loc ad sto"
| "iterateSA loc (s', x) d = (loadRec_Storage (hash loc (ShowL_nat x)) d s', Suc x)"
| "iterateSM loc (k, v) s' = loadRec_Storage (hash loc k) v s'"
```

Environment

datatype *Data_Environment* =

```
    Memarr "Data_Memory list" |
    CDarr "Data_Memory list" |
    Stoarr "Data_Storage list"|
    Stomap "(String.literal × Data_Storage) list" |
    Stackbool bool |
    Stobool bool |
    Stackint int |
    Stoint int |
    Stackaddr Address |
    Stoaddr Address
```

fun *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 *loadsimple_Environment* :: "(Stack × CalldataT × MemoryT × StorageT × Environment)
⇒ (Identifier × Type × Data_Environment) ⇒ (Stack × CalldataT × MemoryT × StorageT × Environment)"
where

```
"loadsimple_Environment (k, c, m, s, e) (id', tp, d) = (case d of
    Stackbool b ⇒
        let (k', e') = astack id' tp (KValue (ShowL_bool b)) (k, e)
        in (k', c, m, s, e')
  | Stobool b ⇒
        let (s', e') = astore id' tp (ShowL_bool b) (s, e)
        in (k, c, m, s', e')
  |  Stackint n ⇒
        let (k', e') = astack id' tp (KValue (ShowL_int n)) (k, e)
        in (k', c, m, s, e')
```

```
  | Stoint n ⇒
        let (s', e') = astore id' tp (ShowL_int n) (s, e)
        in (k, c, m, s', e')
  | Stackaddr ad ⇒
        let (k', e') = astack id' tp (KValue ad) (k, e)
        in (k', c, m, s, e')
  | Stoaddr ad ⇒
        let (s', e') = astore id' tp ad (s, e)
        in (k, c, m, s', e')
  | CDarr a ⇒
        let l = ShowL_nat (toploc c);
            c' = load_Memory a c;
            (k', e') = astack id' tp (KCDptr l) (k, e)
        in (k', c', m, s, e')
  | Memarr a ⇒
        let l = ShowL_nat (toploc m);
            m' = load_Memory a m;
            (k', e') = astack id' tp (KMemptr l) (k, e)
        in (k', c, m', s, e')
  | Stoarr a ⇒
        let s' = loadRec_Storage id' (SArray a) s;
            e' = updateEnv id' tp (Storeloc id') e
        in (k, c, m, s', e')
  | Stomap mp ⇒
        let s' = loadRec_Storage id' (SMap mp) s;
            e' = updateEnv id' tp (Storeloc id') e
        in (k, c, m, s', e')
)"
```

definition getValue_Environment :: "Stack ⇒ CalldataT ⇒ MemoryT ⇒ StorageT ⇒ Environment ⇒ Identifier ⇒ String.literal ⇒ String.literal"
where

```
"getValue_Environment k c m s e i txt = (case fmlookup (denvalue e) i of
    Some (tp, Stackloc l) ⇒ (case accessStore l k of
        Some (KValue v) ⇒ (case tp of
            Value t ⇒ (txt + i + (STR ''=='') + dump_Valuetypes t v + (STR ''↩''))
          |  _ ⇒ FAILURE)
      | Some (KCDptr p) ⇒ (case tp of
            Calldata (MTArray x t) ⇒ dump_Memory p x t c i txt
            | _ ⇒ FAILURE)
      | Some (KMemptr p) ⇒ (case tp of
            Memory (MTArray x t) ⇒ dump_Memory p x t m i txt
            | _ ⇒ FAILURE)
      | Some (KStoptr p) ⇒ (case tp of
            Storage t ⇒ dump_Storage s p i t txt
            | _ ⇒ FAILURE))
   | Some (Storage t, Storeloc l) ⇒ dump_Storage s l i t txt
   | _ ⇒ FAILURE
)"
```

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

Accounts

fun load_Accounts :: "Accounts ⇒ (Address × Balance × atype × nat) list ⇒ Accounts" **where**

```
   "load_Accounts acc []              = acc"
 | "load_Accounts acc ((ad, b, t, c)#as) = load_Accounts (acc (ad:=⦇bal=b, type=Some t, contracts=c⦈)) as"
```

fun dumpStorage::"StorageT ⇒ (Identifier × Member) ⇒ String.literal" **where**

```
  "dumpStorage s (i, Var x) = dump_Storage s i i x EMPTY"
| "dumpStorage s (i, Function x) = FAILURE"
| "dumpStorage s (i, Method x) = FAILURE"
```

```
fun dumpMembers :: "atype option ⇒ Environment_P ⇒ StorageT ⇒ String.literal" where
  "dumpMembers None ep s = FAILURE"
| "dumpMembers (Some EOA) _ _ = STR ''EOA''"
| "dumpMembers (Some (Contract name)) ep s =
    (case ep $$ name of
      Some (ct, _) ⇒ name + STR ''('' + (intersperse (STR '','') (map (dumpStorage s) (filter (is_Var
∘ snd) (sorted_list_of_fmap ct)))) + STR '')''
    | None ⇒ FAILURE)"
```

```
fun dump_Account :: "nat ⇒ Environment_P ⇒ State ⇒ Address ⇒ String.literal"
  where "dump_Account 0 _ _ _ = FAILURE"
  | "dump_Account (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 ''↩'' + dump_Account c ep st (hash a (ShowL_int x))) EMPTY (int (contracts
(accounts st a)))"
```

```
definition dump_Accounts :: "Environment_P ⇒ State ⇒ Address list ⇒ String.literal"
  where "dump_Accounts ep st al = intersperse (STR ''↩'') (map (dump_Account 1000 ep st) al)"
```

```
definition init_Account::"(Address × Balance × atype × nat) list => Accounts" where
  "init_Account = load_Accounts 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, emptyStore, emptyStore, fmempty,
emptyEnv addr name adest aval) dat;
               a        = init_Account 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))
                          + (dump_Accounts 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'',(Memory (MTArray 5 (MTValue TBool)), Memarr [MBool True, MBool False, MBool
True, MBool False, MBool True]))]
       = STR ''v1[0]==true↩v1[1]==false↩v1[2]==true↩v1[3]==false↩v1[4]==true↩089Be5381FcEa58aF3341014
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''), Storage (STArray 1 (STArray 2 (STValue (TSInt 120)))), Stoarr
[SArray [SInt 347104507864064359095275590289383142, SInt 565831699297331399489670920129618233]]),
             ((STR ''a_s120_21_m8''), Memory (MTArray 1 (MTArray 2 (MTValue (TSInt 120)))), Memarr
[MArray [MInt (290845675805142398428016622247257774), MInt ((-968340268772692771706452946692722226))]]))]
  = 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 8 1]) (LVAL (Ref (STR ''a_s8_31_s7'') [UINT 8 0])))
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            EMPTY
            EMPTY
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0)]
            [(STR ''a_s8_31_s7'', (Storage (STArray 1 (STArray 3 (STValue (TSInt 8)))), Stoarr [SArray
[SInt ((98)), SInt ((-23)), SInt (36)]])),
             (STR ''a_s8_32_m0'', (Memory (MTArray 2 (MTArray 3 (MTValue (TSInt 8)))), Memarr [MArray
[MInt ((-64)), MInt ((39)), MInt ((-125))], MArray [MInt ((-32)), MInt ((-82)), MInt ((-105))]]))]
       = STR ''a_s8_32_m0[0][0]==-64 ↩ a_s8_32_m0[0][1]==39 ↩ a_s8_32_m0[0][2]==-125 ↩ a_s8_32_m0[1][0]==98 ↩ a_
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'', (Storage (STMap (TUInt 32) (STValue (TUInt 8))), 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 256 0))
            (STR ''local'')
            EMPTY
            (STR ''mycontract'')
            (STR ''0'')
            [(STR ''local'', STR ''100'', EOA, 0), (STR ''myaddr'', STR ''100'', Contract (STR
''mycontract''), 0)]
            []
        = STR ''local: balance==100 - EOA ↩ myaddr: balance==100 - mycontract(v1==false ↩ )''"
  ⟨proof⟩

lemma "eval1 1000
            (NEW (STR ''mycontract'') [] (UINT 256 10))
```

```
            (STR ''local'')
            EMPTY
            (STR ''mycontract'')
            (STR ''0'')
            [(STR ''local'', STR ''100'', EOA, 0), (STR ''myaddr'', STR ''100'', 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 256 10))
                (EXTERNAL CONTRACTS (STR ''m1'') [] (UINT 256 0))
            )
            (STR ''local'')
            EMPTY
            (STR ''mycontract'')
            (STR ''0'')
            [(STR ''local'', STR ''100'', EOA, 0), (STR ''myaddr'', STR ''100'', 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 8 5))],
              ([], SKIP),
              SKIP)
            fmempty"
```

```
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 8), Stackint 0))]
       = STR ''v1==5↩myaddr: balance==100 - EOA''"
  ⟨proof⟩
```

```
definition "testenv3 ≡ loadProc (STR ''mycontract'')
              ([(STR ''m1'',
                Function ([(STR ''v2'', Value (TSInt 8)), (STR ''v3'', Value (TSInt 8))],
                False,
                PLUS (LVAL (Id (STR ''v2''))) (LVAL (Id (STR ''v3'')))))],
                ([], SKIP),
              SKIP)
            fmempty"
```

```
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 8 3, E.INT 8 4]))
           (STR ''myaddr'')
           (STR ''mycontract'')
           EMPTY
           (STR ''0'')
           [(STR ''myaddr'', STR ''100'', EOA, 0),(STR ''mya'', STR ''100'', EOA, 0)]
           [(STR ''v1'', (Value (TSInt 8), 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 8)), (STR ''v3'', Value (TSInt 8))],
True, PLUS (LVAL (Id (STR ''v2''))) (LVAL (Id (STR ''v3'')))))],
              ([], SKIP),
              SKIP)
           fmempty"

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 8 3, E.INT
8 4]))
           (STR ''myaddr'')
           EMPTY
           EMPTY
           (STR ''0'')
           [(STR ''myaddr'', STR ''100'', EOA, 0), (STR ''extaddr'', STR ''100'', Contract (STR
''mycontract''), 0)]
           [(STR ''v1'', (Value (TSInt 8), Stackint 0))]
       = STR ''v1==7↩myaddr: balance==100 - EOA↩extaddr: balance==100 - mycontract()''"
  ⟨proof⟩

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

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 256 10))
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            EMPTY
            EMPTY
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR ''myaddr'',
STR ''100'', 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)
            fmempty"

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 256 10))
          (STR ''SenderAd'')
          EMPTY
          EMPTY
          (STR ''0'')
          [(STR ''ReceiverAd'', STR ''100'', 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)
             fmempty"

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 8 0))))(TRANSFER (ADDRESS (STR ''myaddr''))
```

```
(UINT 256 5)))(SKIP))
            (STR ''089Be5381FcEa58aF334101414c04F993947C733'')
            EMPTY
            EMPTY
            (STR ''0'')
            [(STR ''089Be5381FcEa58aF334101414c04F993947C733'', STR ''100'', EOA, 0), (STR ''myaddr'',
STR ''100'', Contract (STR ''mycontract''),0)]
            [(STR ''x'', (Value (TUInt 8), Stackint 9))]
       = STR ''x==0↩089Be5381FcEa58aF334101414c04F993947C733: balance==95 - EOA↩myaddr:
balance==105 - mycontract()''"
  ⟨proof⟩
```

6.2.3 The Final Code Export

```
consts ReadL_S   :: "String.literal ⇒ S"
consts ReadL_acc :: "String.literal ⇒ (String.literal × String.literal × atype × nat) list"
consts ReadL_dat :: "String.literal ⇒ (String.literal × Type × Data_Environment) list"
consts ReadL_P :: "String.literal ⇒ Data_P list"

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

fun main_stub :: "String.literal list ⇒ (int × String.literal)"
  where
   "main_stub [stm, saddr, name, raddr, val, acc, dat]
        = (0, eval0 1000 (ReadL_S stm) saddr name raddr val (ReadL_acc acc) (ReadL_dat dat))"
 | "main_stub _ = (2,
          STR ''solidity-evaluator [credit] "Statement" "ContractAddress" "OriginAddress" "Value"↩''
        + STR ''  "(Address * Balance) list" "(Address * ((Identifier * Member) list) * Statement)"
"(Variable * Type * Value) list"↩''
        + STR ''↩'')"

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

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

export_generated_files _

export_code 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 (MTArray 5 (MTArray 6 (MTValue TBool))) (STR ''2'') [UINT 8 3] eempty emptyStore
(mystate⦇gas:=1⦈) 1
= Normal ((STR ''3.2'', MTArray 6 (MTValue TBool)), 1)"  ⟨proof⟩

lemma "msel True (MTArray 5 (MTArray 6 (MTValue TBool))) (STR ''2'') [UINT 8 3, UINT 8 4] eempty
emptyStore (mystate⦇gas:=1,memory:=mymemory2⦈) 1
= Normal ((STR ''4.5'', MTValue TBool), 1)" ⟨proof⟩
```

lemma "msel True (MTArray 5 (MTArray 6 (MTValue TBool))) (STR ''2'') [UINT 8 5] eempty emptyStore (mystate⦇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 (8::nat) 0]) TRUE)"
abbreviation myenv::Environment
  where "myenv ≡ updateEnv (STR ''ma'') (Memory (MTArray 1 (MTValue TBool))) (Stackloc (STR ''1''))
                (updateEnv (STR ''sa'') (Storage (STArray 1 (STValue TBool))) (Storeloc (STR ''1''))
                (emptyEnv (STR ''ad'') EMPTY (STR ''ad'') (STR ''0'')))"

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

abbreviation mystore::"Address ⇒ StorageT"
  where "mystore ≡ λ _ . fmempty"

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

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

declare[[ML_print_depth = 10000]]
value ‹(stmt P1 myenv emptyStore ⦇accounts=emptyAccount, stack=mystack, memory=mymemory,
storage=(mystore ((STR ''ad''):= mystorage)), gas=1000⦈)›


lemma ‹(stmt P1 myenv emptyStore ⦇accounts=emptyAccount, stack=mystack, memory=mymemory,
storage=(mystore ((STR ''ad''):= mystorage)), 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⦈,
storage = (mystore ((STR ''ad''):= mystorage)), gas = 1000⦈) ›
    ⟨proof⟩

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 "⋀v. a = KValue v ⟹ wp (f v) P E s"
      and "⋀p. a = KCDptr p ⟹ wp (g p) P E s"
      and "⋀p. a = KMemptr p ⟹ wp (h p) P E s"
      and "⋀p. a = KStoptr p ⟹ wp (i p) P E s"
    shows "wp (case a of KValue v ⇒ f v | KCDptr p ⇒ g p | KMemptr p ⇒ h p | KStoptr p ⇒ i p) P E
s"
⟨proof⟩

lemma wpmtypes[wprule]:
  assumes "⋀i m. a = MTArray i m ⟹ wp (f i m) P E s"
      and "⋀t. a = MTValue t ⟹ wp (g t) P E s"
    shows "wp (case a of MTArray i m ⇒ f i m | MTValue t ⇒ g t) P E s"
⟨proof⟩

lemma wpstypes[wprule]:
  assumes "⋀i m. a = STArray i m ⟹ wp (f i m) P E s"
      and "⋀t t'. a = STMap t t' ⟹ wp (g t t') P E s"
      and "⋀t. a = STValue t ⟹ wp (h t) P E s"
shows "wp (case a of STArray i m ⇒ f i m | STMap t t' ⇒ g t t' | STValue t ⇒ h t) P E s"
⟨proof⟩

lemma wptype[wprule]:
  assumes "⋀v. a = Value v ⟹ wp (f v) P E s"
      and "⋀m. a = Calldata m ⟹ wp (g m) P E s"
      and "⋀m. a = Memory m ⟹ wp (h m) P E s"
      and "⋀t. a = Storage t ⟹ wp (i t) P E s"
shows "wp (case a of Value v ⇒ f v | Calldata m ⇒ g m | Memory m ⇒ h m | Storage s ⇒ i s) P E s"
  ⟨proof⟩

lemma wptypes[wprule]:
assumes "⋀x. a= TSInt x ⟹ wp (f x) P E s"
    and "⋀x. a = TUInt x ⟹ wp (g x) P E s"
    and "a = TBool ⟹ wp h P E s"
    and "a = TAddr ⟹ wp i P E s"
  shows "wp (case a of TSInt x ⇒ f x | TUInt x ⇒ g x | TBool ⇒ h | TAddr ⇒ i) P E s"
⟨proof⟩

lemma wpltype[wprule]:
  assumes "⋀l. a=LStackloc l ⟹ wp (f l) P E s"
      and "⋀l. a = LMemloc l ⟹ wp (g l) P E s"
      and "⋀l. a = LStoreloc l ⟹ wp (h l) P E s"
    shows "wp (case a of LStackloc l ⇒ f l | LMemloc l ⇒ g l | LStoreloc l ⇒ h l) P E s"
⟨proof⟩

lemma wpdenvalue[wprule]:
  assumes "⋀l. a=Stackloc l ⟹ wp (f l) P E s"
      and "⋀l. a=Storeloc l ⟹ 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

```
type_synonym State_Predicate = "Accounts × Stack × MemoryT × (Address ⇒ StorageT) ⇒ bool"

definition validS :: "State_Predicate ⇒ (unit, Ex ,State) state_monad ⇒ State_Predicate ⇒ (Ex ⇒
bool) ⇒ bool"
  (‹⦃_⦄S/ _ /(⦃_⦄S,/ ⦃_⦄S)›)
where
  "⦃P⦄S f ⦃Q⦄S,⦃E⦄S ≡
    ∀st. P (accounts st, stack st, memory st, storage st)
    ⟶ (case f st of
         Normal (_,st') ⇒ gas st' ≤ gas st ∧ Q (accounts st', stack st', memory st', storage st')
       | Exception e ⇒ E e)"

definition wpS :: "(unit, Ex ,State) state_monad ⇒ (State ⇒ bool) ⇒ (Ex ⇒ bool) ⇒ State ⇒ bool"
  where "wpS f P E st ≡ wp f (λ_ st'. gas st' ≤ gas st ∧ P st') E st"

lemma wpS_valid:
  assumes "⋀st::State. P (accounts st, stack st, memory st, storage st) ⟹ wpS f (λst. Q (accounts
st, stack st, memory st, storage st)) E st"
  shows "⦃P⦄S f ⦃Q⦄S,⦃E⦄S"
  ⟨proof⟩

lemma valid_wpS:
  assumes "⦃P⦄S f ⦃Q⦄S,⦃E⦄S"
  shows "⋀st. P (accounts st, stack st, memory st, storage st) ⟹ wpS f (λst. Q (accounts st, stack
st, memory st, storage st))E st"
  ⟨proof⟩

context statement_with_gas
begin
```

7.2.2 Skip

```
lemma wp_Skip:
  assumes "P (st⦇gas := gas st - costs SKIP ev cd st⦈)"
      and "E Gas"
    shows "wpS (λs. stmt SKIP ev cd s) P E st"
  ⟨proof⟩
```

7.2.3 Assign

```
lemma wp_Assign:
  fixes ex ev cd st lv
  defines "ngas ≡ gas st - costs (ASSIGN lv ex) ev cd st"
  assumes "⋀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');
           g' ≤ gas st;
           convert t t' v = Some v'⟧
         ⟹ P (st⦇gas := g', stack:=updateStore l (KValue v') (stack st)⦈)"
      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 ((LStoreloc l, Storage (STValue t')), g');
           g' ≤ gas st;
           convert t t' v = Some v'⟧
         ⟹ P (st⦇gas := g', storage:=(storage st) (address ev := (fmupd l v' (storage st (address
ev))))⦈)"
      and "⋀v t g l t' g' v' vt.
           ⟦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, Memory (MTValue t')), g');
```

```
        g' ≤ gas st;
        convert t t' v = Some v'⟧
      ⟹ P (st⦇gas := g', memory:=updateStore l (MValue v') (memory st)⦈)"
and "⋀p x t g l t' g' p' m'.
        ⟦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, Memory t'), g');
        g' ≤ gas st;
        accessStore l (stack st) = Some (KMemptr p');
        cpm2m p p' x t cd (memory st) = Some m'⟧
      ⟹ P (st⦇gas := g', memory:=m'⦈)"
and "⋀p x t g l t' g' p' s'.
        ⟦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, Storage t'), g');
        g' ≤ gas st;
        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 "⋀p x t g l t' g' s'.
        ⟦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 ((LStoreloc l, t'), g');
        g' ≤ gas st;
        cpm2s p l x t cd (storage st (address ev)) = Some s'⟧
      ⟹ P (st⦇gas := g', storage:=(storage st) (address ev := s')⦈)"
and "⋀p x t g l t' g' m'.
        ⟦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 ((LMemloc l, t'), g');
        g' ≤ gas st;
        cpm2m p l x t cd (memory st) = Some m'⟧
      ⟹ P (st⦇gas := g', memory:=m'⦈)"
and "⋀p x t g l t' g'.
        ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KMemptr p, Memory (MTArray x t)), g);
        lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStackloc l, Memory t'), g');
        g' ≤ gas st⟧
      ⟹ P (st⦇gas := g', stack:=updateStore l (KMemptr p) (stack st)⦈)"
and "⋀p x t g l t' g' p' s'.
        ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KMemptr p, Memory (MTArray x t)), g);
        lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStackloc l, Storage t'), g');
        g' ≤ gas st;
        accessStore l (stack st) = Some (KStoptr p');
        cpm2s p p' x t (memory st) (storage st (address ev)) = Some s'⟧
      ⟹ P (st⦇gas := g', storage:=(storage st) (address ev := s')⦈)"
and "⋀p x t g l t' g' s'.
        ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KMemptr p, Memory (MTArray x t)), g);
        lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStoreloc l, t'), g');
        g' ≤ gas st;
        cpm2s p l x t (memory st) (storage st (address ev)) = Some s'⟧
      ⟹ P (st⦇gas := g', storage:=(storage st) (address ev := s')⦈)"
and "⋀p x t g l t' g'.
        ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KMemptr p, Memory (MTArray x t)), g);
        lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LMemloc l, t'), g');
        g' ≤ gas st⟧
      ⟹ P (st⦇gas := g', memory:=updateStore l (MPointer p) (memory st)⦈)"
and "⋀p x t g l t' g' p' m'.
        ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KStoptr p, Storage (STArray x t)), g);
        lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStackloc l, Memory t'), g');
        g' ≤ gas st;
        accessStore l (stack st) = Some (KMemptr p');
        cps2m p p' x t (storage st (address ev)) (memory st) = Some m'⟧
      ⟹ P (st⦇gas := g', memory:=m'⦈)"
and "⋀p x t g l t' g'.
        ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KStoptr p, Storage (STArray x t)), g);
        lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStackloc l, Storage t'), g');
        g' ≤ gas st⟧
      ⟹ P (st⦇gas := g', stack:=updateStore l (KStoptr p) (stack st)⦈)"
```

```
      and "⋀p x t g l t' g' s'.
            ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KStoptr p, Storage (STArray x t)), g);
            lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStoreloc l, t'), g');
            g' ≤ gas st;
            copy p l x t (storage st (address ev)) = Some s'⟧
          ⟹ P (st⦇gas := g', storage:=(storage st) (address ev := s')⦈)"
      and "⋀p x t g l t' g' m'.
            ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KStoptr p, Storage (STArray x t)), g);
            lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LMemloc l, t'), g');
            g' ≤ gas st;
            cps2m p l x t (storage st (address ev)) (memory st) = Some m'⟧
          ⟹ P (st⦇gas := g', memory:=m'⦈)"
      and "⋀p t t' g l t'' g'.
            ⟦expr ex ev cd (st⦇gas := ngas⦈) ngas = Normal ((KStoptr p, Storage (STMap t t')), g);
            lexp lv ev cd (st⦇gas := g⦈) g = Normal ((LStackloc l, t''), g');
            g' ≤ gas st⟧
          ⟹ P (st⦇gas := g', stack:=updateStore l (KStoptr p) (stack st)⦈)"
      and "E Gas"
      and "E Err"
    shows "wpS (λs. stmt (ASSIGN lv ex) ev cd s) P E st"
  ⟨proof⟩
```

7.2.4 Composition

```
lemma wp_Comp:
  assumes "wpS (stmt s1 ev cd) (λst. wpS (stmt s2 ev cd) P E st) E (st⦇gas := gas st - costs (COMP s1
s2) ev cd st⦈)"
      and "E Gas"
      and "E Err"
    shows "wpS (λs. stmt (COMP s1 s2) ev cd s) P E st"
  ⟨proof⟩
```

7.2.5 Conditional

```
lemma wp_ITE:
  assumes "⋀g g'. expr ex ev cd (st⦇gas := g⦈) g = Normal ((KValue ⌈True⌉, Value TBool), g') ⟹ wpS
(stmt s1 ev cd) P E (st⦇gas := g'⦈)"
      and "⋀g g'. expr ex ev cd (st⦇gas := g⦈) g = Normal ((KValue ⌈False⌉, Value TBool), g') ⟹ wpS
(stmt s2 ev cd) P E (st⦇gas := g'⦈)"
      and "E Gas"
      and "E Err"
    shows "wpS (λs. stmt (ITE ex s1 s2) ev cd s) P E st"
  ⟨proof⟩
```

7.2.6 While Loop

```
lemma wp_While[rule_format]:
    fixes iv::"Accounts × Stack × MemoryT × (Address ⇒ StorageT) ⇒ bool"
  assumes "⋀a k m s st g. ⟦iv (a, k, m, s); expr ex ev cd (st⦇gas := gas st - costs (WHILE ex sm)
ev cd st⦈) (gas st - costs (WHILE ex sm) ev cd st) = Normal ((KValue ⌈False⌉, Value TBool), g)⟧ ⟹ P
(st⦇gas := g⦈)"
      and "⋀a k m s st g. ⟦iv (a, k, m, s); expr ex ev cd (st⦇gas := gas st - costs (WHILE ex sm) ev
cd st⦈) (gas st - costs (WHILE ex sm) ev cd st) = Normal ((KValue ⌈True⌉, Value TBool), g)⟧ ⟹ wpS
(stmt sm ev cd) (λst. iv (accounts st, stack st, memory st, storage st)) E (st⦇gas:=g⦈)"
      and "E Err"
      and "E Gas"
    shows "iv (accounts st, stack st, memory st, storage st) ⟶ wpS (λs. stmt (WHILE ex sm) ev cd s) P
E st"
⟨proof⟩
```

7.2.7 Blocks

```
lemma wp_blockNone:
```

```
  assumes "⋀cd' mem' sck' e'. decl id0 tp None False cd (memory (st⦇gas := gas st - costs (BLOCK
((id0, tp), None) stm) ev cd st⦈)) (storage (st⦇gas := gas st - costs (BLOCK ((id0, tp), None) stm) ev
cd st⦈))
           (cd, memory (st⦇gas := gas st - costs (BLOCK ((id0, tp), None) stm) ev cd st⦈), stack
(st⦇gas := gas st - costs (BLOCK ((id0, tp), None) stm) ev cd st⦈), ev) = Some (cd', mem', sck', e')
          ⟹ wpS (stmt stm e' cd') P E (st⦇gas := gas st - costs (BLOCK ((id0, tp), None) stm) ev cd
st, stack := sck', memory := mem'⦈)"
      and "E Gas"
      and "E Err"
    shows "wpS (λs. stmt (BLOCK ((id0, tp), None) stm) ev cd s) P E st"
  ⟨proof⟩

lemma wp_blockSome:
  assumes "⋀v t g' cd' mem' sck' e'.
        ⟦ expr ex' ev cd (st⦇gas := gas st - costs (BLOCK ((id0, tp), Some ex') stm) ev cd st⦈) (gas st
- costs (BLOCK ((id0, tp), Some ex') stm) ev cd st) = Normal ((v, t), g');
          g' ≤ gas st - costs (BLOCK ((id0, tp), Some ex') stm) ev cd st;
          decl id0 tp (Some (v,t)) False cd (memory st) (storage st) (cd, memory st, stack st, ev) =
Some (cd', mem', sck', e')⟧
        ⟹ wpS (stmt stm e' cd') P E (st⦇gas := g', stack := sck', memory := mem'⦈)"
      and "E Gas"
      and "E Err"
    shows "wpS (λs. stmt (BLOCK ((id0, tp), Some ex') stm) ev cd s) P E 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 × Type) list × 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 ⇒ int ⇒ bool"
```

7.2.9 Method invocations and transfer

```
definition Qe
  where "Qe ad iv st ≡
    (∀mid fp f ev.
       members $$ mid = Some (Method (fp,True,f)) ∧
       address ev ≠ ad
     ⟶ (∀adex cd st' xe val g v t g' v' e_l cd_l k_l m_l g'' acc.
```

```
            g'' ≤ gas st ∧
            type (acc ad) = Some (Contract cname) ∧
            expr adex ev cd (st'⦇gas := gas st' - costs (EXTERNAL adex mid xe val) ev cd st'⦈) (gas st' - costs (EXTERNAL adex mid xe val) ev cd st') = Normal ((KValue ad, Value TAddr), g) ∧
            expr val ev cd (st'⦇gas := g⦈) g = Normal ((KValue v, Value t), g') ∧
            convert t (TUInt 256) v = Some v' ∧
            load True fp xe (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom members)) emptyStore emptyStore emptyStore ev cd (st'⦇gas := g'⦈) g' = Normal ((e_l, cd_l, k_l, m_l), g'') ∧
            transfer (address ev) ad v' (accounts (st'⦇gas := g''⦈)) = Some acc ∧
            iv (storage st' ad) (ReadL_int (bal (acc ad)) - ReadL_int v')
           ⟶ wpS (stmt f e_l cd_l) (λst. iv (storage st ad) (ReadL_int (bal (accounts st ad)))) (λe. e = Gas ∨ e = Err) (st'⦇gas := g'', accounts:= acc, stack := k_l, memory := m_l⦈)))"
```

definition *Qi*

```
  where "Qi ad pre post st ≡
   (∀mid fp f ev.
     members $$ mid = Some (Method (fp,False,f)) ∧
     address ev = ad
    ⟶ (∀cd st' i xe e_l cd_l k_l m_l g.
          g ≤ gas st ∧
          load False fp xe (ffold (init members) (emptyEnv ad cname (sender ev) (svalue ev)) (fmdom members)) emptyStore emptyStore (memory st') ev cd (st'⦇gas := gas st' - costs (INVOKE i xe) ev cd st'⦈) (gas st' - costs (INVOKE i xe) ev cd st') = Normal ((e_l, cd_l, k_l, m_l), g) ∧
          pre mid (ReadL_int (bal (accounts st' ad)), storage st' ad, e_l, cd_l, k_l, m_l)
         ⟶ wpS (stmt f e_l cd_l) (λst. post mid (ReadL_int (bal (accounts st ad)), storage st ad)) (λe. e = Gas ∨ e = Err) (st'⦇gas := g, stack := k_l, memory := m_l⦈)))"
```

definition *Qfi*

```
  where "Qfi ad pref postf st ≡
   (∀ev. address ev = ad
    ⟶ (∀ex cd st' adex cc v t g g' v' acc.
          g' ≤ gas st ∧
          expr adex ev cd (st'⦇gas := gas st' - cc⦈) (gas st' - cc) = Normal ((KValue ad, Value TAddr), g) ∧
          expr ex ev cd (st'⦇gas := g⦈) g= Normal ((KValue v, Value t), g') ∧
          convert t (TUInt 256) v = Some v' ∧
          transfer (address ev) ad v' (accounts st') = Some acc ∧
          pref (ReadL_int (bal (acc ad)), storage st' ad)
         ⟶ wpS (λs. stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom members)) emptyStore s) (λst. postf (ReadL_int (bal (accounts st ad)), storage st ad)) (λe. e = Gas ∨ e = Err) (st'⦇gas := g', accounts := acc, stack:=emptyStore, memory:=emptyStore⦈)))"
```

definition *Qfe*

```
  where "Qfe ad iv st ≡
   (∀ev. address ev ≠ ad
    ⟶ (∀ex cd st' adex cc v t g g' v' acc.
          g' ≤ gas st ∧
          type (acc ad) = Some (Contract cname) ∧
          expr adex ev cd (st'⦇gas := gas st' - cc⦈) (gas st' - cc) = Normal ((KValue ad, Value TAddr), g) ∧
          expr ex ev cd (st'⦇gas := g⦈) g = Normal ((KValue v, Value t), g') ∧
          convert t (TUInt 256) v = Some v' ∧
          transfer (address ev) ad v' (accounts st') = Some acc ∧
          iv (storage st' ad) (ReadL_int (bal (acc ad)) - ReadL_int v')
         ⟶ wpS (λs. stmt fb (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom members)) emptyStore s) (λst. iv (storage st ad) (ReadL_int (bal (accounts st ad)))) (λe. e = Gas ∨ e = Err) (st'⦇gas := g', accounts := acc, stack:=emptyStore, memory:=emptyStore⦈)))"
```

lemma *safeStore[rule_format]:*

```
  fixes ad iv
  defines "aux1 st ≡ ∀st'::State. gas st' < gas st ⟶ Qe ad iv st'"
      and "aux2 st ≡ ∀st'::State. gas st' < gas st ⟶ Qfe ad iv st'"
    shows "∀st'. address ev ≠ ad ∧ type (accounts st ad) = Some (Contract cname) ∧ iv (storage st ad) (ReadL_int (bal (accounts st ad))) ∧
```

```
        stmt f ev cd st = Normal ((), st') ∧ aux1 st ∧ aux2 st
       ⟶ iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
```
⟨proof⟩

type_synonym *Precondition = "int × StorageT × Environment × Memoryvalue Store × Stackvalue Store × Memoryvalue Store ⇒ bool"*
type_synonym *Postcondition = "int × StorageT ⇒ bool"*

The following lemma can be used to verify (recursive) internal or external method calls and transfers executed from **inside** (`address ev = 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 ≡
    (∀ev ad' i xe val cd.
       address ev = ad ∧
       (∀adv c g v t g' v'.
         expr ad' ev cd (st⦇gas := gas st - costs (EXTERNAL ad' i xe val) ev cd st⦈) (gas st - costs
(EXTERNAL ad' i xe val) ev cd st) = Normal ((KValue adv, Value TAddr), g) ∧
         adv ≠ ad ∧
         type (accounts st adv) = Some (Contract c) ∧
         c |∈| fmdom ep ∧
         expr val ev cd (st⦇gas := g⦈) g = Normal ((KValue v, Value t), g') ∧
         convert t (TUInt 256) v = Some v'
        ⟶ iv (storage st ad) (ReadL_int (bal (accounts st ad)) - ReadL_int v'))
      ⟶ wpS (λs. stmt (EXTERNAL ad' i xe val) ev cd s) (λst. iv (storage st ad) (ReadL_int (bal
(accounts st ad)))) (λe. e = Gas ∨ e = Err) st)"

definition Pi
  where "Pi ad pre post st ≡
    (∀ev i xe cd.
       address ev = ad ∧
       contract ev = cname ∧
       (∀fp e_l cd_l k_l m_l g.
         load False fp xe (ffold (init members) (emptyEnv ad (contract ev) (sender ev) (svalue ev))
(fmdom members)) emptyStore emptyStore (memory st) ev cd (st⦇gas := gas st - costs (INVOKE i xe) ev cd
st⦈) (gas st - costs (INVOKE i xe) ev cd st) = Normal ((e_l, cd_l, k_l, m_l), g)
        ⟶ pre i (ReadL_int (bal (accounts st ad)), storage st ad, e_l, cd_l, k_l, m_l))
    ⟶ wpS (λs. stmt (INVOKE i xe) ev cd s) (λst. post i (ReadL_int (bal (accounts st ad)), storage st
ad)) (λe. e = Gas ∨ e = Err) st)"

definition Pfi
  where "Pfi ad pref postf st ≡
   (∀ev ex ad' cd.
     address ev = ad ∧
     (∀adv g.
       expr ad' ev cd (st⦇gas := gas st - costs (TRANSFER ad' ex) ev cd st⦈) (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 := gas st - costs (TRANSFER ad' ex) ev cd st⦈) (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 (ReadL$_{int}$ (bal (accounts st ad)), storage st ad))
⟶ wpS (λs. stmt (TRANSFER ad' ex) ev cd s) (λst. postf (ReadL$_{int}$ (bal (accounts st ad)), storage st ad)) (λe. e = Gas ∨ e = Err) st)"

definition *Pfe*
where *"Pfe ad iv st ≡*
(∀ ev ex ad' cd.
address ev = ad ∧
(∀ adv g.
expr ad' ev cd (st⦇gas := gas st - costs (TRANSFER ad' ex) ev cd st⦈) (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 := gas st - costs (TRANSFER ad' ex) ev cd st⦈) (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 256) v = Some v'
⟶ iv (storage st ad) (ReadL$_{int}$ (bal (accounts st ad)) - ReadL$_{int}$ v'))
⟶ wpS (λs. stmt (TRANSFER ad' ex) ev cd s) (λst. iv (storage st ad) (ReadL$_{int}$ (bal (accounts st ad)))) (λe. e = Gas ∨ e = Err) st)"

lemma *wp_external_invoke_transfer:*
fixes *pre::"Identifier ⇒ Precondition"*
and *post::"Identifier ⇒ Postcondition"*
and *pref::"Postcondition"*
and *postf::"Postcondition"*
and *iv::"Invariant"*
assumes *assm: "⋀st::State.*
⟦∀ st'::State. gas st' ≤ gas st ∧ type (accounts st' ad) = Some (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 (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 (Contract cname)"*
and *"⋀st::State. ⟦∀ st'::State. gas st' ≤ gas st ∧ type (accounts st' ad) = Some (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 := gas st - costs (TRANSFER ad' ex) ev cd st⦈) (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 := gas st - costs (TRANSFER ad' ex) ev cd st⦈) (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 256) v = Some v'
⟶ iv (storage st ad) (ReadL$_{int}$ (bal (accounts st ad)) - ReadL$_{int}$ v'))
⟶ wpS (λs. stmt (TRANSFER ad' ex) ev cd s) (λst. iv (storage st ad) (ReadL$_{int}$ (bal (accounts st ad)))) (λe. e = Gas ∨ e = Err) st)"
⟨*proof*⟩

lemma *wp_external[rule_format]:*
assumes *"type (accounts st ad) = Some (Contract cname)"*

```
     and "⋀st::State. ⟦∀st'::State. gas st' ≤ gas st ∧ type (accounts st' ad) = Some (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 ad' i xe val cd.
       address ev = ad ∧
       (∀adv c g v t g' v'.
         expr ad' ev cd (st⦇gas := gas st - costs (EXTERNAL ad' i xe val) ev cd st⦈) (gas st - costs
(EXTERNAL ad' i xe val) ev cd st) = Normal ((KValue adv, Value TAddr), g) ∧
         adv ≠ ad ∧
         type (accounts st adv) = Some (Contract c) ∧
         c |∈| fmdom ep ∧
         expr val ev cd (st⦇gas := g⦈) g = Normal ((KValue v, Value t), g') ∧
         convert t (TUInt 256) v = Some v'
        ⟶ iv (storage st ad) (ReadL_int (bal (accounts st ad)) - ReadL_int v'))
      ⟶ wpS (λs. stmt (EXTERNAL ad' i xe val) ev cd s) (λst. iv (storage st ad) (ReadL_int (bal
(accounts st ad)))) (λe. e = Gas ∨ e = Err) st)"
```
⟨proof⟩

```
lemma wp_invoke[rule_format]:
  assumes "type (accounts st ad) = Some (Contract cname)"
      and "⋀st::State. ⟦∀st'::State. gas st' ≤ gas st ∧ type (accounts st' ad) = Some (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 i xe cd.
       address ev = ad ∧
       contract ev = cname ∧
       (∀fp e_l cd_l k_l m_l g.
         load False fp xe (ffold (init members) (emptyEnv ad (contract ev) (sender ev) (svalue ev))
(fmdom members)) emptyStore emptyStore (memory st) ev cd (st⦇gas := gas st - costs (INVOKE i xe) ev cd
st⦈) (gas st - costs (INVOKE i xe) ev cd st) = Normal ((e_l, cd_l, k_l, m_l), g)
        ⟶ pre i (ReadL_int (bal (accounts st ad)), storage st ad, e_l, cd_l, k_l, m_l))
    ⟶ wpS (λs. stmt (INVOKE i xe) ev cd s) (λst. post i (ReadL_int (bal (accounts st ad)), storage st
ad)) (λe. e = Gas ∨ e = Err) st)"
```
⟨proof⟩

```
lemma wp_transfer_int[rule_format]:
  assumes "type (accounts st ad) = Some (Contract cname)"
      and "⋀st::State. ⟦∀st'::State. gas st' ≤ gas st ∧ type (accounts st' ad) = Some (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 := gas st - costs (TRANSFER ad' ex) ev cd st⦈) (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 := gas st - costs (TRANSFER ad' ex) ev cd st⦈) (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 (ReadL_int (bal (accounts st ad)), storage st ad))
    ⟶ wpS (λs. stmt (TRANSFER ad' ex) ev cd s) (λst. postf (ReadL_int (bal (accounts st ad)), storage
st ad)) (λe. e = Gas ∨ e = Err) st)"
```
⟨proof⟩

```
definition constructor :: "((String.literal, String.literal) fmap ⇒ int ⇒ bool) ⇒ bool"
  where "constructor iv ≡ (∀acc g'' m_l k_l cd_l e_l g' t v xe i cd val st ev adv.
    adv = hash (address ev) (ShowL_nat (contracts (accounts st (address ev)))) ∧
    type (accounts st adv) = None ∧
    expr val ev cd (st⦇gas := gas st - costs (NEW i xe val) ev cd st⦈) (gas st - costs (NEW i xe val)
ev cd st) = Normal ((KValue v, Value t), g') ∧
    load True (fst const) xe (ffold (init members) (emptyEnv adv cname (address ev) v) (fmdom members))
emptyStore emptyStore emptyStore ev cd (st⦇gas := g'⦈) g' = Normal ((e_l, cd_l, k_l, m_l), g'') ∧
    transfer (address ev) adv v (accounts (st⦇accounts := (accounts st)(adv := ⦇bal = ShowL_int 0, type =
```

```
Some (Contract i), contracts = 0⦈)⦈)) = Some acc
    ⟶ wpS (local.stmt (snd const) e_l cd_l) (λst. iv (storage st adv) ⌈bal (accounts st adv)⌉) (λe. e =
Gas ∨ e = Err)
        (st⦇gas := g'', storage:=(storage st)(adv := {$$}), accounts := acc, stack:=k_l, memory:=m_l⦈))"

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 (Contract cname) ⟶ iv (storage st ad) (ReadL_int (bal
(accounts st ad)))"
    shows "∀(st'::State). stmt f ev cd st = Normal ((), st') ∧ type (accounts st' ad) = Some (Contract
cname)
            ⟶ iv (storage st' ad) (ReadL_int (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. address ev ≠ ad ∧ type (accounts st ad) = Some (Contract cname) ⟶ iv (storage st
ad) (ReadL_int (bal (accounts st ad)))"
    shows "∀(st'::State) ad. stmt f ev cd st = Normal ((), st') ∧ type (accounts st' ad) = Some
(Contract cname) ∧ address ev ≠ ad
            ⟶ iv (storage st' ad) (ReadL_int (bal (accounts st' ad)))"
  ⟨proof⟩
end

context Calculus
begin

  named_theorems mcontract
  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

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

The weakest precondition for assignments generates a lot of different cases. However, usually only one of them is required for a given situation.

The following rule eliminates the wrong cases by proving that they lead to a contradiction. It requires two facts to be provided:

- `expr_rule`: should be a theorem which evaluates the expression part of the assignment
- `lexp_rule`: should be a theorem which evaluates the left hand side of the assignment

Both theorems should assume a corresponding loading of parameters and all declarations which happen before the assignment.

```
method vcg_insert_expr_lexp for ex::E and lv::L uses expr_rule lexp_rule =
  match premises in
    expr: "expr ex _ _ _ _ = _" and
    lexp: "lexp lv _ _ _ _ = _" ⇒
      ‹insert expr_rule[OF expr] lexp_rule[OF lexp]›

method vcg_insert_decl for ex::E and lv::L uses expr_rule lexp_rule =
  match premises in
    decl: "decl _ _ _ _ _ _ _ _ = _" (multi) ⇒
      ‹vcg_insert_expr_lexp ex lv expr_rule:expr_rule[OF decl] lexp_rule:lexp_rule[OF decl]›
  | _ ⇒
      ‹vcg_insert_expr_lexp ex lv expr_rule:expr_rule lexp_rule:lexp_rule›

method vcg_insert_load for ex::E and lv::L uses expr_rule lexp_rule =
  match premises in
    load: "load _ _ _ _ _ _ _ _ _ _ _ = _" ⇒
      ‹vcg_insert_decl ex lv expr_rule:expr_rule[OF load] lexp_rule:lexp_rule[OF load]›
  | _ ⇒
      ‹vcg_insert_decl ex lv expr_rule:expr_rule lexp_rule:lexp_rule›

method vcg_assign uses expr_rule lexp_rule =
  match conclusion in
    "wpS (stmt (ASSIGN lv ex) _ _) _ _ _" for lv ex ⇒
      ‹rule wp_Assign;
        (solve ‹(rule FalseE, simp,
          (vcg_insert_load ex lv expr_rule:expr_rule lexp_rule:lexp_rule)), simp›
       | solve simp)?›,
      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 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:

- *pref::int × (String.literal, String.literal) fmap ⇒ bool*: Precondition for fallback method called internally
- *postf::int × (String.literal, String.literal) fmap ⇒ bool*: Postcondition for fallback method called internally
- *pre::String.literal ⇒ int × (String.literal, String.literal) fmap × Environment × Memoryvalue Store × Stackvalue Store × Memoryvalue Store ⇒ bool*: Preconditions for internal methods
- *post::String.literal ⇒ int × (String.literal, String.literal) fmap ⇒ 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*.
- *cases_ext*: performs case distinction over *external* methods of contract *ad*.
- *cases_int*: performs case distinction over *internal* methods of contract *ad*.
- *cases_fb*: performs case distinction over *fallback* method of 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;
        (vcg_prep,
         rule external;
           solve ‹assumption | simp›)›
  | "Qi _ _ _ _" ⇒
      ‹unfold Qi_def,
       vcg_prep,
       erule cases_fb;
        (vcg_prep,
         rule internal;
           solve ‹assumption | simp›)›
  | "Qfi _ _ _ _" ⇒
      ‹unfold Qfi_def,
       rule allI,
       rule impI,
       rule cases_int;
        (vcg_prep,
```

```
        rule fallback_int;
          solve ‹assumption | simp›)›
  | "Qfe _ _ _" ⇒
      ‹unfold Qfe_def,
       rule allI,
       rule impI,
       rule cases_int;
        (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],*

```
  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
  "HOL-Eisbach.Eisbach_Tools"
begin
```

8.1.1 Example of Re-entrancy

```
definition[solidity_symbex]:"example_env ≡
             loadProc (STR ''Attacker'')
               ([],
               ([], SKIP),
                ITE (LESS (BALANCE THIS) (UINT 256 125))
                  (EXTERNAL (ADDRESS (STR ''BankAddress'')) (STR ''withdraw'') [] (UINT 256 0))
                  SKIP)
             (loadProc (STR ''Bank'')
               ([(STR ''balance'', Var (STMap TAddr (STValue (TUInt 256)))),
               (STR ''deposit'', Method ([], True,
                 ASSIGN
                   (Ref (STR ''balance'') [SENDER])
                   (PLUS (LVAL (Ref (STR ''balance'') [SENDER])) VALUE))),
               (STR ''withdraw'', Method ([], True,
                 ITE (LESS (UINT 256 0) (LVAL (Ref (STR ''balance'') [SENDER])))
                   (COMP
```

```
                    (TRANSFER SENDER (LVAL (Ref (STR ''balance'') [SENDER])))
                    (ASSIGN (Ref (STR ''balance'') [SENDER]) (UINT 256 0)))
                  SKIP))],
              ([], 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 256 10))
            (EXTERNAL (ADDRESS (STR ''BankAddress'')) (STR ''withdraw'') [] (UINT 256 0)))
          (STR ''AttackerAddress'')
          (STR ''Attacker'')
          (STR '''')
          (STR ''0'')
          [(STR ''BankAddress'', STR ''100'', Contract (STR ''Bank''), 0), (STR ''AttackerAddress'', STR
''100'', 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 ≡ Ref (STR ''balance'') [SENDER]"*

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

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

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

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

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

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

abbreviation *"banklist ≡ [*
```
          (STR ''balance'', Var (STMap TAddr (STValue (TUInt 256)))),
          (STR ''deposit'', Method ([], True, deposit)),
          (STR ''withdraw'', Method ([], True, keep))]"
```

definition *bank::"(Identifier, Member) fmap"*
where *"bank ≡ 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[mcontract]:
  "members $$ STR ''withdraw'' = Some (Method ([], True, keep))"
  ⟨proof⟩

lemma mdeposit[mcontract]:
  "members $$ STR ''deposit'' = Some (Method ([], True, deposit))"
  ⟨proof⟩
```

Variable lemma

```
lemma balance:
  "members $$ (STR ''balance'') = Some (Var (STMap TAddr (STValue (TUInt 256))))"
  ⟨proof⟩
```

Case lemmas

These lemmas are required by *vcg_transfer*.

```
lemma cases_ext:
  assumes "members $$ mid = Some (Method (fp,True,f))"
      and "fp = [] ⟹ P deposit"
      and "fp = [] ⟹ P keep"
    shows "P f"
⟨proof⟩

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

lemma cases_fb:
  assumes "P SKIP"
  shows "P fb"
⟨proof⟩

lemma cases_cons:
  assumes "fst const = [] ⟹ P (fst const, SKIP)"
  shows "P const"
⟨proof⟩
```

Definition of Invariant

```
abbreviation "SUMM s ≡ ∑(ad,x)|fmlookup s (ad + (STR ''.'' + STR ''balance'')) = Some x. ReadL_int x"

abbreviation "POS s ≡ ∀ad x. fmlookup s (ad + (STR ''.'' + STR ''balance'')) = Some x ⟶ ReadL_int x
≥ 0"

definition "iv s a ≡ a ≥ SUMM s ∧ POS s"

lemma weaken:
  assumes "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v)"
      and "ReadL_int v ≥0"
    shows "iv (storage st ad) (ReadL_int (bal (acc ad)))"
⟨proof⟩
```

Additional lemmas

lemma *expr_0:*
  assumes *"load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)) emptyStore emptyStore emptyStore env cd (st⦇gas := g1⦈) g1 = Normal ((e_l, cd_l, k_l, m_l), g1')"*
      and *"decl STR ''bal'' (Value (TUInt 256)) (Some (lv, lt)) False cd_l m_l s (cd_l, m_l, k_l, e_l) = Some (cd', mem', sck', e')"*
      and *"expr (UINT 256 0) ev0 cd0 st0 g0 = Normal ((rv, rt),g''a)"*
    shows *"rv= KValue ($ShowL_{int}$ 0)"* **and** *"rt=Value (TUInt 256)"*
  ⟨*proof*⟩

lemma *load_empty_par:*
  assumes *"load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)) emptyStore emptyStore emptyStore env cd (st⦇gas := g1⦈) g1 = Normal ((e_l, cd_l, k_l, m_l), g1')"*
    shows *"load True [] [] (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)) emptyStore emptyStore emptyStore env cd (st⦇gas := g1⦈) g1 = Normal ((e_l, cd_l, k_l, m_l), g1')"*
⟨*proof*⟩

lemma *lexp_myrexp_decl:*
  assumes *"load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)) emptyStore emptyStore emptyStore env cd (st⦇gas := g1⦈) g1 = Normal ((e_l, cd_l, k_l, m_l), g1')"*
      and *"decl STR ''bal'' (Value (TUInt 256)) (Some (lv, lt)) False cd_l m_l s (cd_l, m_l, k_l, e_l) = 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=Storage (STValue (TUInt 256))"*
⟨*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 STR ''bal'' (Value (TUInt 256)) (KValue (accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance'')) s')) (k_l, e_l)"*
    shows *"(⌈accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance'')) s'⌉::int) = $ReadL_{int}$ lv"* (**is** *?G1*) **and** *"t = TUInt 256"*
⟨*proof*⟩

lemma *lexp_myrexp:*
  assumes *"load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)) emptyStore emptyStore emptyStore env cd (st⦇gas := g1⦈) g1 = Normal ((e_l, cd_l, k_l, m_l), g1')"*
      and *"lexp myrexp e_l cd_l (st'⦇gas := g2⦈) g2 = Normal ((rv,rt), g2')"*
    shows *"rv= LStoreloc (address env + (STR ''.'' + STR ''balance''))"* **and** *"rt=Storage (STValue (TUInt 256))"*
⟨*proof*⟩

lemma *expr_balance:*
  assumes *"load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members)) emptyStore emptyStore emptyStore env cd (st⦇gas := g1⦈) g1 = Normal ((e_l, cd_l, k_l, m_l), g1')"*
      and *"expr (LVAL (Ref (STR ''balance'') [SENDER])) e_l cd_l (st⦇accounts := acc, stack := k_l, memory := m_l, gas := g2⦈) g2 = Normal ((va, ta), g'a)"*
    shows *"va= KValue (accessStorage (TUInt 256) (address env + (STR ''.'' + STR ''balance'')) (storage st ad))"*
      and *"ta = Value (TUInt 256)"*
⟨*proof*⟩

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

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

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

lemma expr_plus:
  assumes "load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members))
emptyStore emptyStore emptyStore env cd (st⦇gas := g3⦈) g3 = Normal ((e_l, cd_l, k_l, m_l), g3')"
      and "expr (PLUS a0 b0) ev0 cd0 st0 g0 = Normal ((xs, t'0), g'0)"
    shows "∃s. xs = KValue (s)"
  ⟨proof⟩

lemma summ_eq_sum:
  "SUMM s' = (∑(ad,x)|fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr. ReadL_int
x)
           + ReadL_int (accessStorage (TUInt 256) (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. ReadL_int
x)
          =(∑(ad,x)|fmlookup s' (ad + (STR ''.'' + STR ''balance'')) = Some x ∧ ad ≠ adr. ReadL_int
x)"
⟨proof⟩

lemma adapt_deposit:
  assumes "address env ≠ ad"
      and "load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members))
emptyStore emptyStore emptyStore env cd (st0⦇gas := g3⦈) g3 = Normal ((e_l, cd_l, k_l, m_l), g3')"
      and "Accounts.transfer (address env) ad v a = Some acc"
      and "iv (storage st0 ad) (ReadL_int (bal (acc ad)) - ReadL_int v)"
      and "lexp myrexp e_l cd_l (st0⦇gas := g'', accounts := acc, stack := k_l, memory := m_l, gas := g'⦈)
g' = Normal ((LStoreloc l, Storage (STValue t')), g''a)"
      and "expr (PLUS mylval VALUE) e_l cd_l (st0⦇gas := g'', accounts := acc, stack := k_l, memory := m_l,
gas := g⦈) g = Normal ((KValue va, Value ta), g')"
      and "Valuetypes.convert ta t' va = Some v'"
    shows "(ad = address e_l ⟶ iv (storage st0 (address e_l)(l $$:= v')) ⌈bal (acc (address e_l))⌉) ∧
           (ad ≠ address e_l ⟶ iv (storage st0 ad) (ReadL_int (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')) (l $$:= v'))⦈"
  assumes "iv (storage st ad) (ReadL_int (bal (acc ad)) - ReadL_int v)"
      and "load True [] xe (ffold (init members) (emptyEnv ad cname (address env) v) (fmdom members))
emptyStore
           emptyStore emptyStore env cd (st⦇gas := g'⦈) g' = Normal ((e_l, cd_l, k_l, m_l), g'')"
      and "decl STR ''bal'' (Value (TUInt 256)) (Some (va, ta)) False cd_l m_l (storage st) (cd_l, m_l, k_l,
e_l) =
           Some (cd', mem', sck', e')"
      and "expr (UINT 256 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, 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 "expr SENDER e' cd' (st' ⦇gas := g⦈) g =  Normal ((KValue adv, Value TAddr), g'x)"
      and adv_def: "adv ≠ ad"
      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 256) lv = Some lv'"
    shows "iv (storage st' ad) (ReadL_int (bal (accounts st' ad)) - (ReadL_int lv'))"
⟨proof⟩

lemma wp_deposit[external]:
  assumes "address ev ≠ ad"
      and "expr adex ev cd (st⦇gas := gas st0 - costs (EXTERNAL adex mid xe val) ev cd st0⦈) (gas st0 -
costs (EXTERNAL adex mid xe val) ev cd st0) = Normal ((KValue ad, Value TAddr), g)"
      and "expr val ev cd (st0⦇gas := g⦈) g = Normal ((KValue v, Value t), g')"
      and "Valuetypes.convert t (TUInt 256) v = Some v'"
      and "load True [] xe (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom members))
emptyStore emptyStore emptyStore ev cd (st0⦇gas := g'⦈) g' = Normal ((e_l, cd_l, k_l, m_l), g'')"
      and "Accounts.transfer (address ev) ad v' (accounts st0) = Some acc"
      and "iv (storage st0 ad) (ReadL_int (bal (acc ad)) - ReadL_int v')"
    shows "wpS (stmt (ASSIGN myrexp (PLUS mylval VALUE)) e_l cd_l)
           (λst. (iv (storage st ad) (ReadL_int (bal (accounts st ad))))) (λe. e = Gas ∨ e = Err)
           (st0⦇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 "g'' ≤ gas st"
      and "type (acc ad) = Some (Contract cname)"
      and "expr adex ev cd (st0⦇gas := gas st0 - costs (EXTERNAL adex mid xe val) ev cd st0⦈)
           (gas st0 - costs (EXTERNAL adex mid xe val) ev cd st0) =
           Normal ((KValue ad, Value TAddr), g)"
      and "expr val ev cd (st0⦇gas := g⦈) g = Normal ((KValue gv, Value gt), g')"
      and "Valuetypes.convert gt (TUInt 256) gv = Some gv'"
      and "load True [] xe (ffold (init members) (emptyEnv ad cname (address ev) gv') (fmdom members))
emptyStore
           emptyStore emptyStore 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 256)) (Some (v, t)) False cd_l m_l (storage st0)
           (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, 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 256)), Some mylval)
           (COMP (ASSIGN myrexp (UINT 256 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 256)), Some mylval) (COMP (ASSIGN myrexp
(UINT 256 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 256 0) e' cd' (st0⦇accounts := acc, stack := sck', memory := mem', gas := ga⦈) ga
= Normal ((KValue va, Value ta), g'a)"
    shows "wpS (stmt Reentrancy.transfer e' cd') (λst. iv (storage st ad) (ReadL_int (bal (accounts st
ad))))
        (λe. e = Gas ∨ e = Err) st'"
⟨proof⟩

lemma wp_withdraw[external]:
```

assumes "⋀st'. gas st' ≤ gas st ∧ type (accounts st' ad) = Some (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'' ≤ gas st"
and "type (acc ad) = Some (Contract cname)"
and "expr adex ev cd (st0⦇gas := gas st0 - costs (EXTERNAL adex mid xe val) ev cd st0⦈) (gas st0 - costs (EXTERNAL adex mid xe val) ev cd st0) = Normal ((KValue ad, Value TAddr), g)"
and "expr val ev cd (st0⦇gas := g⦈) g = Normal ((KValue v, Value t), g')"
and "Valuetypes.convert t (TUInt 256) v = Some v'"
and "load True [] xe (ffold (init members) (emptyEnv ad cname (address ev) v') (fmdom members)) emptyStore emptyStore emptyStore ev cd (st0⦇gas := g'⦈) g' = Normal ((e_l, cd_l, k_l, m_l), g'')"
and "Accounts.transfer (address ev) ad v' (accounts st0) = Some acc"
and "iv (storage st0 ad) (ReadL$_{int}$ (bal (acc ad)) - ReadL$_{int}$ v')"
shows "wpS (stmt keep e_l cd_l)
(λst. iv (storage st ad) (ReadL$_{int}$ (bal (accounts st ad)))) (λe. e = Gas ∨ e = Err)
(st0⦇gas := g'', accounts := acc, stack := k_l, memory := m_l⦈)"
⟨*proof*⟩

lemma *wp_fallback:*
assumes "Accounts.transfer (address ev) ad v (accounts st0) = Some acca"
and "iv (storage st0 ad) (ReadL$_{int}$ (bal (acca ad)) - ReadL$_{int}$ v)"
shows "wpS (stmt SKIP (ffold (init members) (emptyEnv ad cname (address ev) vc) (fmdom members)) emptyStore)
(λst. iv (storage st ad) (ReadL$_{int}$ (bal (accounts st ad)))) (λe. e = Gas ∨ e = Err)
(st0⦇gas := g'c, accounts := acca, stack := emptyStore, memory := emptyStore⦈)"
⟨*proof*⟩

lemma *wp_construct:*
assumes "Accounts.transfer (address ev) (hash (address ev) ⌊contracts (accounts st (address ev))⌋) v
((accounts st) (hash (address ev) ⌊contracts (accounts st (address ev))⌋ := ⦇bal = ShowL$_{int}$ 0, type = Some (Contract i), contracts = 0⦈)) =
Some acc"
shows "iv fmempty ⌈bal (acc (hash (address ev) ⌊contracts (accounts st (address ev))⌋))⌉"
⟨*proof*⟩

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

Final results

interpretation *vcg:VCG costs$_e$ ep costs cname members const fb "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. address ev ≠ ad ∧ type (accounts st ad) = Some (Contract cname) ⟶ iv (storage st ad) (ReadL$_{int}$ (bal (accounts st ad)))"
shows "∀(st'::State) ad. stmt f ev cd st = Normal ((), st') ∧ type (accounts st' ad) = Some (Contract cname) ∧ address ev ≠ ad
⟶ iv (storage st' ad) (ReadL$_{int}$ (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 (b∈vbits)
      then if v ≥ 0
        then E.INT b (-(2^(b-1)) + (v+2^(b-1)) mod (2^b))
        else E.INT b (2^(b-1) - (-v+2^(b-1)-1) mod (2^b) - 1)
      else E.INT b v)"
| "eupdate (UINT b v) = (if (b∈vbits) then UINT b (v mod (2^b)) else UINT b v)"
| "eupdate (ADDRESS a) = ADDRESS a"
| "eupdate (BALANCE a) = BALANCE a"
| "eupdate THIS = THIS"
| "eupdate SENDER = SENDER"
| "eupdate VALUE = VALUE"
| "eupdate TRUE = TRUE"
| "eupdate FALSE = FALSE"
| "eupdate (LVAL l) = LVAL (lupdate l)"
| "eupdate (PLUS ex1 ex2) =
    (case (eupdate ex1) of
      E.INT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            E.INT b2 v2 ⇒
              if b2∈vbits
                then let v=v1+v2 in
                  if v ≥ 0
                    then E.INT (max b1 b2) (-(2^((max b1 b2)-1)) + (v+2^((max b1 b2)-1)) mod (2^(max b1
b2)))
                    else E.INT (max b1 b2) (2^((max b1 b2)-1) - (-v+2^((max b1 b2)-1)-1) mod (2^(max b1
b2)) - 1)
                else (PLUS (E.INT b1 v1) (E.INT b2 v2))
          | UINT b2 v2 ⇒
              if b2∈vbits ∧ b2 < b1
                then let v=v1+v2 in
                  if v ≥ 0
                    then E.INT b1 (-(2^(b1-1)) + (v+2^(b1-1)) mod (2^b1))
                    else E.INT b1 (2^(b1-1) - (-v+2^(b1-1)-1) mod (2^b1) - 1)
                else PLUS (E.INT b1 v1) (UINT b2 v2)
          | _ ⇒ PLUS (E.INT b1 v1) (eupdate ex2))
        else PLUS (E.INT b1 v1) (eupdate ex2)
    | UINT b1 v1 ⇒
        if b1 ∈ vbits
          then (case (eupdate ex2) of
            UINT b2 v2 ⇒
              if b2 ∈ vbits
                then UINT (max b1 b2) ((v1 + v2) mod (2^(max b1 b2)))
                else (PLUS (UINT b1 v1) (UINT b2 v2))
```

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

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

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

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

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

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

lemma no_gas:
  assumes "¬ 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.